《最少必要面试题》第一版

相信大家都会有种及眼熟又陌生的感觉、看过可能在短暂的面试后又马上忘记了。**JavaPub** 在这里整理这些容易忘记的重点知识及 **解答**，`建议收藏，经常温习查阅`。

[点击在线阅读《最少必要面试题》]

作者：JavaPub

2024

缓存

1. 什么是缓存?
2. 为什么要用缓存?
3. 请说说有哪些缓存算法？是否能手写一下 LRU 代码的实现?
4. 常见的常见的缓存工具和框架有哪些?
5. 用了缓存之后，有哪些常见问题?
6. 如何处理缓存穿透的问题
7. 如何处理缓存雪崩的问题
8. 如何处理缓存击穿的问题
9. 缓存和 DB 的一致性如何保证?
10. 什么是缓存预热？如何实现缓存预热?

拓展：缓存数据的淘汰策略有哪些?

Docker

1. 什么是 Docker 容器?
2. Docker 和虚拟机有什么不同?
3. 什么是 DockerFile?
4. 使用Docker Compose时如何保证容器A先于容器B运行?
5. 一个完整的Docker由哪些部分组成?
6. docker常用命令
7. 描述 Docker 容器的生命周期。
8. docker容器之间怎么隔离?

ElasticSearch

1. 说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段 。
2. elasticsearch 的倒排索引是什么
3. elasticsearch 是如何实现 master 选举的
5. 描述一下 Elasticsearch 索引文档的过程
4. 详细描述一下 Elasticsearch 搜索的过程?
5. Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法
6. Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办?
7. 客户端在和集群连接时，如何选择特定的节点执行请求的?
8. 详细描述一下 Elasticsearch 更新和删除文档的过程。
9. Elasticsearch 对于大数据量（上亿量级）的聚合如何实现?
10. 在并发情况下，Elasticsearch 如果保证读写一致?
11. 介绍一下你们的个性化搜索方案?

Java基础

1. instanceof 关键字的作用
2. Java自动装箱和拆箱
3. 重载和重写区别
4. equals与==区别
5. 谈谈NIO和BIO区别
6. String、StringBuffer、StringBuilder 的区别是什么?
7. 泛型是什么，有什么特点
8. final 有哪些用法
9. 说一下Java注解
10. Java创建对象有几种方式

Java并发

1. start()方法和run()方法的区别
2. volatile关键字的作用
3. sleep方法和wait方法有什么区别
4. 如何停止一个正在运行的线程?
5. java如何实现多线程之间的通讯和协作?（如何在两个线程间共享数据？）

缓存

1. 什么是缓存?

缓存，就是数据交换的缓冲区，针对服务对象的不同（本质就是不同的硬件）都可以构建缓存。而我们平时说的缓存，大多是指内存。

目的是， **把读写速度【慢】的介质的数据保存在读写速度【快】的介质中，从而提高读写速度，减少时间消耗。** 例如：

- CPU 高速缓存 ： 高速缓存的读写速度远高于内存。
  - CPU 读数据时，如果在高速缓存中找到所需数据，就不需要读内存
  - CPU 写数据时，先写到高速缓存，再回写到内存。
- 磁盘缓存：磁盘缓存其实就把常用的磁盘数据保存在内存中，内存读写速度也是远高于磁盘的。
  - 读数据，时从内存读取。
  - 写数据时，可先写到内存，定时或定量回写到磁盘，或者是同步回写。

2. 为什么要用缓存?

使用缓存的目的，就是提升读写性能。而实际业务场景下，更多的是为了提升读性能，带来更好的性能，更高的并发量。

日常业务中，我们使用比较多的数据库是 MySQL ，缓存是 Redis 。Redis 比 MySQL 的读写性能好很多。那么，我们将 MySQL 的热点数据，缓存到 Redis 中，提升读取性能，也减小 MySQL 的读取压力。例如说：

- 论坛帖子的访问频率比较高，且要实时更新阅读量，使用 Redis 记录帖子的阅读量，可以提升性能和并发。
- 商品信息，数据更新的频率不高，但是读取的频率很高，特别是热门商品。

3. 请说说有哪些缓存算法？是否能手写一下 LRU 代码的实现?

缓存算法，比较常见的是三种：

- LRU（least recently used ， 最近最少使用)
- LFU（Least Frequently used ， 最不经常使用)
- FIFO（first in first out ， 先进先出)

这里我们可以借助 LinkedHashMap 实现

```
public class LRULinkedMap<K,V> {


    /**
     * 最大缓存大小
     */
    private int cacheSize;
```

```
    private LinkedHashMap<K,V> cacheMap ;


    public LRULinkedMap(int cacheSize) {
        this.cacheSize = cacheSize;

        cacheMap = new LinkedHashMap(16,0.75F,true){
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                if (cacheSize + 1 == cacheMap.size()){
                    return true ;
                }else {
                    return false ;
                }
            }
        };
    }

    public void put(K key,V value){
        cacheMap.put(key,value) ;
    }

    public V get(K key){
        return cacheMap.get(key) ;
    }


    public Collection<Map.Entry<K, V>> getAll() {
        return new ArrayList<Map.Entry<K, V>>(cacheMap.entrySet());
    }
}
```

使用案例:

```
    @Test
    public void put() throws Exception {
        LRULinkedMap<String,Integer> map = new LRULinkedMap(3) ;
        map.put("1",1);
        map.put("2",2);
        map.put("3",3);

        for (Map.Entry<String, Integer> e : map.getAll()){
            System.out.print(e.getKey() + " : " + e.getValue() + "\t");
        }

        System.out.println("");
        map.put("4",4);
        for (Map.Entry<String, Integer> e : map.getAll()){
            System.out.print(e.getKey() + " : " + e.getValue() + "\t");
        }
    }

//输出
1 : 1   2 : 2   3 : 3
2 : 2   3 : 3   4 : 4
```

4. 常见的常见的缓存工具和框架有哪些?

在 Java 后端开发中，常见的缓存工具和框架列举如下：

- 本地缓存：Guava LocalCache、Ehcache、Caffeine 。

  Ehcache 的功能更加丰富，Caffeine 的性能要比 Guava LocalCache 好。
- 分布式缓存：Redis、Memcached、Tair 。

  Redis 最为主流和常用。

5. 用了缓存之后，有哪些常见问题?

常见的问题，可列举如下：

写入问题

- 缓存何时写入？并且写时如何避免并发重复写入?
- 缓存如何失效?
- 缓存和 DB 的一致性如何保证?

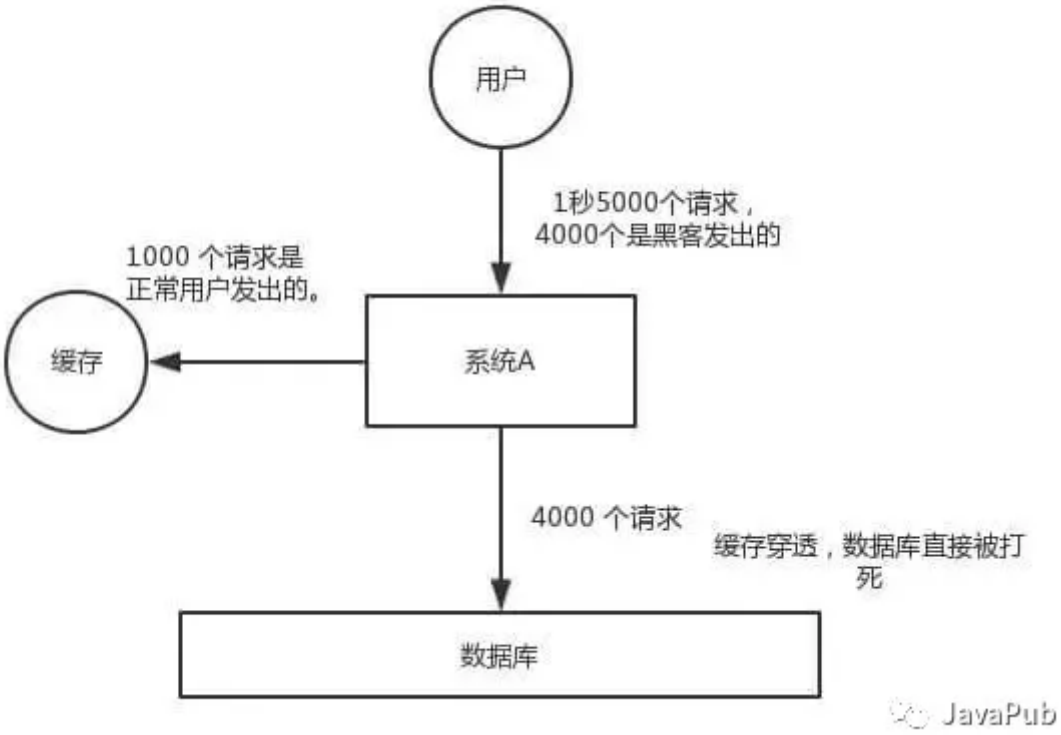
经典三连问

- 如何避免缓存穿透的问题?
- 如何避免缓存击穿的问题?
- 如果避免缓存雪崩的问题?

6. 如何处理缓存穿透的问题

缓存穿透，是指查询一个一定不存在的数据，由于缓存是不命中时被动写，并且处于容错考虑，如果从 DB 查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到 DB 去查询，失去了缓存的意义。

在流量大时，可能 DB 就挂掉了，要是有人利用不存在的 key 频繁攻击我们的应用，这就是漏洞。如下图：



如何解决

有两种方案可以解决：

1. 方案一，缓存空对象。
   当从 DB 查询数据为空，我们仍然将这个空结果进行缓存，具体的值需要使用特殊的标识，能和真正缓存的数据区分开。另外，需要设置较短的过期时间，一般建议不要超过 5 分钟。
2. 方案二，BloomFilter 布隆过滤器。
   在缓存服务的基础上，构建 BloomFilter 数据结构，在 BloomFilter 中存储对应的 KEY 是否存在，如果存在，说明该 KEY 对应的值不为空。

如何选择

这两个方案，各有其优缺点。

	缓存空对象	**BloomFilter 布隆过滤器**
适用场景	1、数据命中不高 2、保证一致性	1、数据命中不高, 2、数据相对固定、实时性低
维护成本	1、代码维护简单 2、需要过多的缓存空间 3、数据不一致	1、代码维护复杂，2、缓存空间占用小

实际情况下，使用方案二比较多。因为，相比方案一来说，更加节省内容，对缓存的负荷更小。

7. 如何处理缓存雪崩的问题

缓存雪崩，是指缓存由于某些原因无法提供服务(例如，缓存挂掉了)，所有请求全部达到 DB 中，导致 DB 负荷大增，最终挂掉的情况。

如何解决

预防和解决缓存雪崩的问题，可以从以下多个方面进行共同着手。

1. 缓存高可用：通过搭建缓存的高可用，避免缓存挂掉导致无法提供服务的情况，从而降低出现缓存雪崩的情况。假设我们使用 Redis 作为缓存，则可以使用 Redis Sentinel 或 Redis Cluster 实现高可用。
2. 本地缓存：如果使用本地缓存时，即使分布式缓存挂了，也可以将 DB 查询到的结果缓存到本地，避免后续请求全部到达 DB 中。如果我们使用 JVM ，则可以使用 Ehcache、Guava Cache 实现本地缓存的功能。

> 当然，引入本地缓存也会有相应的问题，例如说：
>
> 本地缓存的实时性怎么保证?
> 方案一，可以引入消息队列。在数据更新时，发布数据更新的消息；而进>程中有相应的消费者消费该消息，从而更新本地缓存。
> 方案二，设置较短的过期时间，请求时从 DB 重新拉取。
> 方案三，手动过期。

3. 请求 DB 限流: 通过限制 DB 的每秒请求数，避免把 DB 也打挂了。如果我们使用 Java ，则可以使用 Guava RateLimiter、Sentinel、Hystrix 实现限流的功能。这样至少能有两个好处：
    - 可能有一部分用户，还可以使用，系统还没死透。
    - 未来缓存服务恢复后，系统立即就已经恢复，无需再处理 DB 也挂掉的情况。
4. 提前演练：在项目上线前，演练缓存宕掉后，应用以及后端的负载情况以及可能出现的问题，在此基础上做一些预案设定。

8. 如何处理缓存击穿的问题

缓存击穿，是指某个极度“热点”数据在某个时间点过期时，恰好在这个时间点对这个 KEY 有大量的并发请求过来，这些请求发现缓存过期一般都会从 DB 加载数据并回设到缓存，但是这个时候大并发的请求可能会瞬间 DB 压垮。

- 对于一些设置了过期时间的 KEY ，如果这些 KEY 可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑这个问题。
- 区别：
    - 和缓存“雪崩”“”的区别在于，前者针对某一 KEY 缓存，后者则是很多 KEY 。
    - 和缓存“穿透”“”的区别在于，这个 KEY 是真实存在对应的值的。

如何解决

有两种方案可以解决：

1. 方案一，使用互斥锁。请求发现缓存不存在后，去查询 DB 前，使用分布式锁，保证有且只有一个线程去查询 DB ，并更新到缓存。
2. 方案二，手动过期。缓存上从不设置过期时间，功能上将过期时间存在 KEY 对应的 VALUE 里。流程如下：

1. 获取缓存。通过 VALUE 的过期时间，判断是否过期。如果未过期，则直接返回；如果已过期，继续往下执行。
2. 通过一个后台的异步线程进行缓存的构建，也就是“手动”过期。通过后台的异步线程，保证有且只有一个线程去查询 DB。
3. 同时，虽然 VALUE 已经过期，还是直接返回。通过这样的方式，保证服务的可用性，虽然损失了一定的时效性。

选择

这两个方案，各有其优缺点。

	使用互斥锁	**手动过期**
优点	1、思路简单 2、保证一致性	1、性价最佳，用户无需等待
缺点	1、代码复杂度增大 2、存在死锁的风险	1、无法保证缓存一致性

9. 缓存和 DB 的一致性如何保证?

产生原因

主要有两种情况，会导致缓存和 DB 的一致性问题：

1. 并发的场景下，导致读取老的 DB 数据，更新到缓存中。

> 主要指的是，更新 DB 数据之前，先删除 Cache 的数据。在低并发量下没什么问题，但是在高并发下，就会存在问题。在(删除 Cache 的数据, 和更新 DB 数据)时间之间，恰好有一个请求，我们如果使用被动读，因为此时 DB 数据还是老的，又会将老的数据写入到 Cache 中。

2. 缓存和 DB 的操作，不在一个事务中，可能只有一个 DB 操作成功，而另一个 Cache 操作失败，导致不一致。

当然，有一点我们要注意，缓存和 DB 的一致性，我们指的更多的是最终一致性。我们使用缓存只要是提高读操作的性能，真正在写操作的业务逻辑，还是以数据库为准。例如说，我们可能缓存用户钱包的余额在缓存中，在前端查询钱包余额时，读取缓存，在使用钱包余额时，读取数据库。

解决方案

在开始说解决方案之前，胖友先看看如下几篇文章，可能有一丢丢多，保持耐心。

当然无论哪种方案，比较重要的就是解决两个问题：

- 1. 将缓存可能存在的并行写，实现串行写。
- 2. 实现数据的最终一致性。

1. 先淘汰缓存，再写数据库
   因为先淘汰缓存，所以数据的最终一致性是可以得到有效的保证的。为什么呢？先淘汰缓存，即使写数据库发生异常，也就是下次缓存读取时，多读取一次数据库。

那么，我们需要解决缓存并行写，实现串行写。比较简单的方式，引入分布式锁。

- 在写请求时，先淘汰缓存之前，先获取该分布式锁。
- 在读请求时，发现缓存不存在时，先获取分布式锁。

2. 先写数据库，再更新缓存

按照 “先写数据库，再更新缓存”，我们要保证 DB 和缓存的操作，能够在 “同一个事务”中，从而实现最终一致性

10. 什么是缓存预热？如何实现缓存预热？

缓存预热

在刚启动的缓存系统中，如果缓存中没有任何数据，如果依靠用户请求的方式重建缓存数据，那么对数据库的压力非常大，而且系统的性能开销也是巨大的。

此时，最好的策略是启动时就把热点数据加载好。这样，用户请求时，直接读取的就是缓存的数据，而无需去读取 DB 重建缓存数据。举个例子，热门的或者推荐的商品，需要提前预热到缓存中。

如何实现

一般来说，有如下几种方式来实现：

- 数据量不大时，项目启动时，自动进行初始化。
- 写个修复数据脚本，手动执行该脚本。
- 写个管理界面，可以手动点击，预热对应的数据到缓存中。

拓展：缓存数据的淘汰策略有哪些？

除了缓存服务器自带的缓存自动失效策略之外，我们还可以根据具体的业务需求进行自定义的“手动”缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存。
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！Redis 的缓存淘汰策略就是很好的实践方式。

具体用哪种方案，大家可以根据自己的应用场景来权衡。

Docker

1. 什么是 Docker 容器？

Docker 是一种流行的开源软件平台，可简化创建、管理、运行和分发应用程序的过程。它使用容器来打包应用程序及其依赖项。我们也可以将容器视为 Docker 镜像的运行时实例。

2. Docker 和虚拟机有什么不同？

Docker 是轻量级的沙盒，在其中运行的只是应用，虚拟机里面还有额外的系统。

3. 什么是 DockerFile？

Dockerfile 是一个文本文件，其中包含我们需要运行以构建 Docker 镜像的所有命令，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。Docker 使用 Dockerfile 中的指令自动构建镜像。我们可以 `docker build` 用来创建按顺序执行多个命令行指令的自动构建。

一些最常用的指令如下：

FROM ：使用 FROM 为后续的指令建立基础映像。在所有有效的 Dockerfile 中， FROM 是第一条指令。

LABEL： LABEL 指令用于组织项目映像，模块，许可等。在自动化布署方面 LABEL 也有很大用途。在 LABEL 中指定一组键值对，可用于程序化配置或布署 Docker 。

RUN： RUN 指令可在映像当前层执行任何命令并创建一个新层，用于在映像层中添加功能层，也许最来的层会依赖它。

CMD： 使用 CMD 指令为执行的容器提供默认值。在 Dockerfile 文件中，若添加多个 CMD 指令，只有最后的 CMD 指令运行。

4. 使用Docker Compose时如何保证容器A先于容器B运行?

> Docker Compose 是一个用来定义和运行复杂应用的Docker工具。一个使用Docker容器的应用，通常由多个容器组成。使用Docker Compose不再需要使用shell脚本来启动容器。Compose 通过一个配置文件来管理多个Docker容器。简单理解：Docker Compose 是docker的管理工具。

Docker Compose 在继续下一个容器之前不会等待容器准备就绪。为了控制我们的执行顺序，我们可以使用“**取决于**”条件，`depends_on` 。这是在 docker-compose.yml 文件中使用的示例

```
version: "2.4"

services:

 backend:

   build: .    # 构建自定义镜像

   depends_on:

     - db

 db:

   image: mysql
```

用 `docker-compose up` 命令将按照我们指定的依赖顺序启动和运行服务。

5. 一个完整的Docker由哪些部分组成?

- DockerClient 客户端
- Docker Daemon 守护进程
- Docker Image 镜像
- DockerContainer 容器

6. docker常用命令

> 命令建议在本地安装做一个实操，记忆会更深刻。
> 也可以克隆基于docker的俩万（springboot+vue）项目练手，提供视频+完善文档。地址：https://gitee.com/rodert/liawan-vue

1. 查看本地主机的所用镜像：`docker images``
2. 搜索镜像：`docker search mysql``
3. 下载镜像：`docker pull mysql`，没写 tag 就默认下载最新的 lastest
4. 下载指定版本的镜像：`docker pull mysql:5.7``

5. 删除镜像：`docker rmi -f 镜像id 镜像id 镜像id``

7. 描述 Docker 容器的生命周期。

Docker 容器经历以下阶段：

- 创建容器
- 运行容器
- 暂停容器（可选）
- 取消暂停容器（可选）
- 启动容器
- 停止容器
- 重启容器
- 杀死容器
- 销毁容器

8. docker容器之间怎么隔离?

> 这是一道涉猎很广泛的题目，理解性阅读。

Linux中的PID、IPC、网络等资源是全局的，而Linux的NameSpace机制是一种资源隔离方案，在该机制下这些资源就不再是全局的了，而是属于某个特定的NameSpace，各个NameSpace下的资源互不干扰。

Namespace实际上修改了应用进程看待整个计算机“视图”，即它的“视线”被操作系统做了限制，只能“看到”某些指定的内容。对于宿主机来说，这些被“隔离”了的进程跟其他进程并没有区别。

虽然有了NameSpace技术可以实现资源隔离，但进程还是可以不受控的访问系统资源，比如CPU、内存、磁盘、网络等，为了控制容器中进程对资源的访问，Docker采用control groups技术(也就是cgroup)，有了cgroup就可以控制容器中进程对系统资源的消耗了，比如你可以限制某个容器使用内存的上限、可以在哪些CPU上运行等等。

有了这两项技术，容器看起来就真的像是独立的操作系统了。

> 强烈建议大家实操，才能更好的理解docker。

低谷蓄力

ElasticSearch

1. 说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段 。

节点数、分片数、副本数，尽量根据自己公司使用情况回答，当然适当放大也可行。

调优手段是现在很常见的面试题，下面这几种调优手段一定要了解懂。当然，下面的每一条都可以当做调优的一部分。

设计调优

参考：
https://www.cnblogs.com/sanduzxcvbnm/p/12084012.html

a. 根据业务增量需求，采取基于日期模板创建索引，通过 `rollover API` 滚动索引；(rollover API我会单独写一个代码案例做讲解，公众号：JavaPub)
b. 使用别名进行索引管理；（es的索引名不能改变，提供的别名机制使用非常广泛。）
c. 每天凌晨定时对索引做force_merge操作，以释放空间；
d. 采取冷热分离机制，热数据存储到SSD，提高检索效率；冷数据定期进行shrink操作，以缩减存储；

e. 采取curator进行索引的生命周期管理；
f. 仅针对需要分词的字段，合理的设置分词器；
g. Mapping阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。

进100+原创文章：https://gitee.com/rodert/JavaPub

写入调优

1. 写入前副本数设置为0；
2. 写入前关闭refresh_interval设置为-1，禁用刷新机制；
3. 写入过程中：采取bulk批量写入；
4. 写入后恢复副本数和刷新间隔；
5. 尽量使用自动生成的id。

查询调优

1. 禁用wildcard；（通配符模式，类似于%like%）
2. 禁用批量terms（成百上千的场景）；
3. 充分利用倒排索引机制，能keyword类型尽量keyword；
4. 数据量大时候，可以先基于时间敲定索引再检索；
5. 设置合理的路由机制。

2. elasticsearch 的倒排索引是什么

倒排索引也就是单词到文档的映射，当然不只是存里文档id这么简单。还包括：词频（TF，Term Frequency）、偏移量（offset）、位置（Posting）。

3. elasticsearch 是如何实现 master 选举的

> ElasticSearch 的选主是 ZenDiscovery 模块负责，源码分析将首发在。 https://gitee.com/rodert/JavaPub

1. 对所有可以成为 Master 的节点（node.master: true）根据 nodeId 排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第0位）节点，暂且认为它是 Master 节点。
2. 如果对某个节点的投票数达到一定的值（可以成为master节点数n/2+1）并且该节点自己也选举自己，那这个节点就是master。否则重新选举。
   (当然也可以自己设定一个值，最小值设定为超过能成为Master节点的n/2+1，否则会出现脑裂问题。discovery.zen.minimum_master_nodes)

5. 描述一下 Elasticsearch 索引文档的过程

1. 客户端向 Node 1 发送新建、索引或者删除请求。
2. 节点使用文档的 _id 确定文档属于分片 0 。请求会被转发到 Node 3，因为分片 0 的主分片目前被分配在 Node 3 上。
3. Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

一图胜千文，记住这幅图，上面是文档在节点间分发的过程，接着说一下文档从接收到写入磁盘过程。
协调节点默认使用文档 ID 参与计算（也支持通过 routing），以便为路由提供合适的分片。

> shard = hash(document_id) % (num_of_primary_shards)

1. 当分片所在的节点接收到来自协调节点的请求后，会将请求写入到 MemoryBuffer，然后定时（默认是每隔 1 秒）写入到 Filesystem Cache，这个从 MomeryBuffer 到 Filesystem Cache 的过程就叫做 refresh；

2. 当然在某些情况下，存在 Momery Buffer 和 Filesystem Cache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystem cache 中的数据写入到磁盘中时，才会清除掉，这个过程叫做 flush；
3. 在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的 translog 将被删除并开始一个新的 translog。
4. flush 触发的时机是定时触发（默认 30 分钟）或者 translog 变得太大（默认为 512M）时；

```
1. translog 可以理解为就是一个文件，一直追加。
2. MemoryBuffer 应用缓存。
3. Filesystem Cache 系统缓冲区。
```

延伸阅读：Lucene 的 `Segement`：

> 1. Lucene 索引是由多个段组成，段本身是一个功能齐全的倒排索引。
> 2. 段是不可变的，允许 Lucene 将新的文档增量地添加到索引中，而不用从头重建索引。
> 3. 对于每一个搜索请求而言，索引中的所有段都会被搜索，并且每个段会消耗CPU 的时钟周、文件句柄和内存。这意味着段的数量越多，搜索性能会越低。
> 4. 为了解决这个问题，Elasticsearch 会合并小段到一个较大的段，提交新的合并段到磁盘，并删除那些旧的小段。

4. 详细描述一下 Elasticsearch 搜索的过程?

> es作为一个分布式的存储和检索系统，每个文档根据 _id 字段做路由分发被转发到对应的shard上。

搜索执行阶段过程分俩个部分，我们称之为 Query Then Fetch。

4.1 query-查询阶段

当一个search请求发出的时候，这个query会被广播到索引里面的每一个shard（主shard或副本shard），每个shard会在本地执行查询请求后会生成一个命中文档的优先级队列。

这个队列是一个排序好的top N数据的列表，它的size等于from+size的和，也就是说如果你的from是10，size是10，那么这个队列的size就是20，所以这也是为什么深度分页不能用from+size这种方式，因为from越大，性能就越低。

es里面分布式search的查询流程如下：

查询阶段包含以下三个步骤:

1. 客户端发送一个 search 请求到 Node 3， Node 3 会创建一个大小为 from + size 的空优先队列。
2. Node 3 将查询请求转发到索引的每个主分片或副本分片中。每个分片在本地执行查询并添加结果到大小为 from + size 的本地有序优先队列中。
3. 每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，也就是 Node 3，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

4.2 fetch - 读取阶段 / 取回阶段

分布式阶段由以下步骤构成：

1. 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。
2. 每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。
3. 一旦所有的文档都被取回了，协调节点返回结果给客户端。

协调节点首先决定哪些文档 确实 需要被取回。例如，如果我们的查询指定了 { "from": 90, "size": 10 } ，最初的90个结果会被丢弃，只有从第91个开始的10个结果需要被取回。这些文档可能来自和最初搜索请求有关的一个、多个甚至全部分片。

协调节点给持有相关文档的每个分片创建一个 multi-get request ，并发送请求给同样处理查询阶段的分片副本。

分片加载文档体-- _source 字段—如果有需要，用元数据和 search snippet highlighting 丰富结果文档。 一旦协调节点接收到所有的结果文档，它就组装这些结果为单个响应返回给客户端。

```
拓展阅读：
深翻页（Deep Pagination）
---
先查后取的过程支持用 from 和 size 参数分页，但是这是 有限制的 。 要记住需要传递信息给协调节点的每个分片必须先创建一个 from + size 长度的队列，协调节点需要根据 number_of_shards * (from + size) 排序文档，来找到被包含在 size 里的文档。

取决于你的文档的大小，分片的数量和你使用的硬件，给 10,000 到 50,000 的结果文档深分页（ 1,000 到 5,000 页）是完全可行的。但是使用足够大的 from 值，排序过程可能会变得非常沉重，使用大量的 CPU、内存和带宽。因为这个原因，我们强烈建议你不要使用深分页。

实际上， “深分页”很少符合人的行为。当2到3页过去以后，人会停止翻页，并且改变搜索标准。会不知疲倦地一页一页的获取网页直到你的服务崩溃的罪魁祸首一般是机器人或者web spider。

如果你 确实 需要从你的集群取回大量的文档，你可以通过用 scroll 查询禁用排序使这个取回行为更有效率，我们会在 later in this chapter 进行讨论。
注：https://www.elastic.co/guide/cn/elasticsearch/guide/current/scroll.html
```

5. Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法

1. 关闭缓存swap;

> 原因：大多数操作系统会将内存使用到文件系统缓存，会将应用程序未用到的内存交换出去。会导致jvm的堆内存交换到磁盘上。交换会导致性能问题。会导致内存垃圾回收延长。会导致集群节点响应时间变慢，或者从集群中断开。

2. 堆内存设置为：Min（节点内存/2, 32GB）；
3. 设置最大文件句柄数；

后俩点不懂可以先说有一定了解，关注JavaPub会做详细讲解。

4. 调整线程池和队列大小
5. 磁盘存储 raid 方式——存储有条件使用 RAID6，增加单节点性能以及避免单节点存储故障。

> https://www.elastic.co/cn/blog/how-to-design-your-elasticsearch-data-storage-architecture-for-scale#raid56

6. Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办?

1. 当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量（discovery.zen.minimum_master_nodes）超过所有候选节点一半以上来解决脑裂问题；
2. 当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data节点，避免脑裂问题。

7. 客户端在和集群连接时，如何选择特定的节点执行请求的?

client 远程连接连接一个 elasticsearch 集群。它并不加入到集群中，只是获得一个或者多个初始化的地址，并以轮询的方式与这些地址进行通信。

8. 详细描述一下 Elasticsearch 更新和删除文档的过程。

1. 删除和更新也都是写操作，但是 Elasticsearch 中的文档是不可变的，因此不能被删除或者改动以展示其变更；(根本原因是底层lucene的segment段文件不可更新删除)
2. 磁盘上的每个段都有一个相应的 .del 文件。当删除请求发送后，文档并没有真 的被删除，而是在 `.del` 文件中被标记为删除。该文档依然能匹配查询，但是会在 结果中被过滤掉。当段合并时，在.del 文件中被标记为删除的文档将不会被写入 新段。
3. 在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新 时，旧版本的文档在.del 文件中被标记为删除，新版本的文档被索引到一个新段。

旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

9. Elasticsearch 对于大数据量（上亿量级）的聚合如何实现?

> 这道题目较难，相信大家看到很多类似这种回答

Elasticsearch 提供的首个近似聚合是cardinality 度量。它提供一个字段的基数，即该字段的distinct或者unique值的数目。它是基于HLL算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 = 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。

科普&拓展:

```
HyperLogLog：
下面简称为HLL，它是 LogLog 算法的升级版，作用是能够提供不精确的去重计数。存在以下的特点：
1. 能够使用极少的内存来统计巨量的数据，在 Redis 中实现的 HyperLogLog，只需要12K内存就能统计2^64个数据。
2. 计数存在一定的误差，误差率整体较低。标准误差为 0.81% 。
3. 误差可以被设置辅助计算因子进行降低。
---
应用场景：
1. 基数不大，数据量不大就用不上，会有点大材小用浪费空间
2. 有局限性，就是只能统计基数数量，而没办法去知道具体的内容是什么
3. 和bitmap相比，属于两种特定统计情况，简单来说，HyperLogLog 去重比 bitmap 方便很多
4. 一般可以bitmap和hyperloglog配合使用，bitmap标识哪些用户活跃，hyperloglog计数
---
应用场景：
1. 基数不大，数据量不大就用不上，会有点大材小用浪费空间
2. 有局限性，就是只能统计基数数量，而没办法去知道具体的内容是什么
3. 和bitmap相比，属于两种特定统计情况，简单来说，HyperLogLog 去重比 bitmap 方便很多
4. 一般可以bitmap和hyperloglog配合使用，bitmap标识哪些用户活跃，hyperloglog计数
来源：刷刷面试
```

10. 在并发情况下，Elasticsearch 如果保证读写一致?

> 首先要了解什么是一致性，在分布式系统中，我们一般通过CPA理论分析。

分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容忍性（P：Partition Tolerance），最多只能同时满足其中两项。

1. 可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；
2. 另外对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。
3. 对于读操作，可以设置 replication 为 sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置 replication 为 async 时，也可以通过设置搜索请求参数_preference 为 primary 来查询主分片，确保文档是最新版本。

11. 介绍一下你们的个性化搜索方案?

> 如果你没有很多实战经验，可以基于 word2vec 做一些练习，我的博客提供了 word2vec Java版的一些Demo。

基于 word2vec 和 Elasticsearch 实现个性化搜索，它有以下优点：

1. 基于word2vec的商品向量还有一个可用之处，就是可以用来实现相似商品的推荐；

Java基础

1. instanceof 关键字的作用

instanceof 是 Java 的保留关键字。它的作用是测试它左边的对象是否是它右边的类的实例，返回 boolean 的数据类型。

```
boolean result = obj instanceof class
```

当 obj 为 Class 的对象，或者是其直接或间接子类，或者是其接口的实现类，结果result 都返回 true，否则返回false。

注意一点：编译器会检查 obj 是否能转换成右边的class类型，如果不能转换则直接报错，如果不能确定类型，则通过编译，具体看运行时定。

obj 必须为引用类型，只能作为对象的判断，不能是基本类型。

```
int i = 0;
System.out.println(i instanceof Integer);//编译不通过
System.out.println(i instanceof Object);//编译不通过
```

源码参考：JavaSE 8 instanceof 的实现算法：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.instanceof

2. Java自动装箱和拆箱

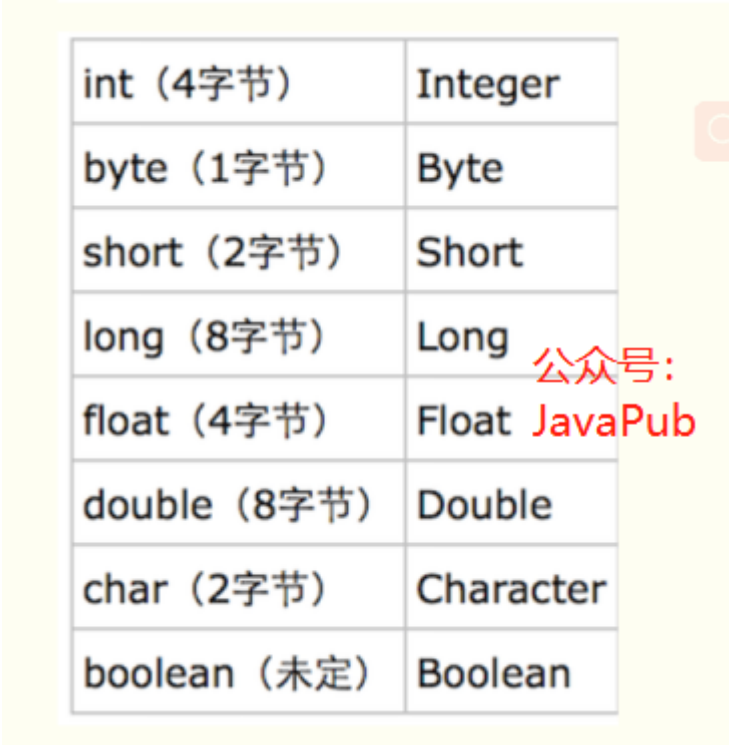
什么是装箱拆箱，这里不做源码层面解读，源码解读在JavaPub公众号发出。这里通过讲解 int 和 Interger 区别，解答Java自动装箱和拆箱。

> 自动装箱 ----- 基本类型的值 → 包装类的实例

> 自动拆箱 ----- 基本类型的值 ← 包装类的实例

1. Integer变量必须实例化后才能使用，而int变量不需要
2. Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值 。
3. Integer的默认值是null，int的默认值是0

Java中8种基本数据类型。左边基本类型，右边包装类型。



在面试中：

下面这段代码的输出结果是什么？

```
public class Main {
    public static void main(String[] args) {

        Integer i1 = 100;
        Integer i2 = 100;
        Integer i3 = 200;
        Integer i4 = 200;

        System.out.println(i1==i2);
        System.out.println(i3==i4);
    }
}

//true
//false
```

输出结果表明i1和i2指向的是同一个对象，而i3和i4指向的是不同的对象。此时只需一看源码便知究竟，下面这段代码是Integer的valueOf方法的具体实现：

```
public static Integer valueOf(int i) {
        if(i >= -128 && i <= IntegerCache.high)
            return IntegerCache.cache[i + 128];
        else
            return new Integer(i);
}
```

```
private static class IntegerCache {
        static final int high;
        static final Integer cache[];

        static {
            final int low = -128;

            // high value may be configured by property
            int h = 127;
            if (integerCacheHighPropValue != null) {
                // Use Long.decode here to avoid invoking methods that
                // require Integer's autoboxing cache to be initialized
                int i = Long.decode(integerCacheHighPropValue).intValue();
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - -low);
            }
            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);
        }

        private IntegerCache() {}
    }
```

> 从这2段代码可以看出，在通过**valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。**

上面的代码中i1和i2的数值为100，因此会直接从cache中取已经存在的对象，所以i1和i2指向的是同一个对象，而i3和i4则是分别指向不同的对象。

`注意，Integer、Short、Byte、Character、Long这几个类的valueOf方法的实现是类似的。`

`Double、Float的valueOf方法的实现是类似的（没有缓存数值，这里的数值想想都有很多，不适合缓存）。`

---

3. 重载和重写区别

重载和重写是一个特别好理解的概念，这里说一个通俗的解答方式

重载（Overload）:首先是位于一个类之中或者其子类中，具有相同的方法名，但是方法的参数不同，返回值类型可以相同也可以不同。

1. 方法名必须相同
2. 方法的参数列表一定不一样。
3. 访问修饰符和返回值类型可以相同也可以不同。

其实简单而言：重载就是对于不同的情况写不同的方法。 比如，同一个类中，写不同的构造函数用于初始化不同的参数。

```
public class JavaPubTest {
    public void out(){
        System.out.println("参数"+null);
    }
    //参数数目不同
    public void out(Integer n){
        System.out.println("参数"+n.getClass().getName());
    }

    //参数类型不同
    public void out(String string){
        System.out.println("参数"+string.getClass().getName());
    }

    public void out(Integer n ,String string){
        System.out.println("参
数"+n.getClass().getName()+","+string.getClass().getName());
    }
    //参数顺序不同
    public void out(String string,Integer n){
        System.out.println("参
数"+string.getClass().getName()+","+n.getClass().getName());
    }

    public static void main(String[] args) {
        JavaPubTest javaPubTest = new JavaPubTest();
        javaPubTest.out();
        javaPubTest.out(1);
        javaPubTest.out("string");
        javaPubTest.out(1,"string");
        javaPubTest.out("string",1);
    }
}
```

重写（Overriding）发生在父类子类之间，比如所有类都是继承与Object类的，Object类中本身就有equals、hashcode、toString方法等。在任意子类中定义了重名和同样的参数列表就构成方法重写。

1. 方法名必须相同，返回值类型必须相同。
2. 参数列表必须相同。
3. 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
4. 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。
5. 构造方法不能被重写。

---

4. equals与==区别

"=="是判断两个变量或实例是不是指向同一个内存空间。
"equals"是判断两个变量或实例所指向的内存空间的值是不是相同。

除了这俩点，这个问题大概率会引出以下问题：

> 为什么重写equals还要重写hashcode?

通过上面俩条我们知道 **"equals"是判断两个变量或实例所指向的内存空间的值是不是相同。** 但是一些特殊场景，我们需要对比俩个对象是否相等，例如：`User user1 = new User();` `User user2 = new User();` user1 和 user2 对比。这是我们就需要重写 equals 方法。

所以可以通过重写equals()方法来判断对象的值是否相等，但是有一个要求：**equals()方法实现了等价关系**，即：

- 自反性：对于任何非空引用x，x.equals(x)应该返回true；
- 对称性：对于任何引用x和y，如果x.equals(y)返回true，那么y.equals(x)也应该返回true；
- 传递性：对于任何引用x、y和z，如果x.equals(y)返回true，y.equals(z)返回true，那么x.equals(z)也应该返回true；
- 一致性：如果x和y引用的对象没有发生变化，那么反复调用x.equals(y)应该返回同样的结果；
- 非空性：对于任意非空引用x，x.equals(null)应该返回false；

到这里也是一个很正常的操作，但是当我们要用到 HashSet 等集合时。存储的对象我们需要用`hashcode 判断对象是否存在`，如果使用 Object 默认的hashcode方法，那我们同样属性的俩个用户一定是不相等的(例如下面user3、user4)，因为内存地址不同，这并不符合我们的业务，所以决定了重写 hashcode 的必要性。

```
User user3 = new User("JavaPub", "man", "1996-08-28")
User user4 = new User("JavaPub", "man", "1996-08-28")
```

---

5. 谈谈NIO和BIO区别

致力于大白话说清楚。NIO和BIO是一个相对有点抽象的概念，如果你对网络有点了解，理解起来可能会更顺畅。首先说一下基本

BIO：同步阻塞IO，每一个客户端连接，服务端都会对应一个处理线程，对于没有分配到处理线程的连接就会被阻塞或者拒绝。相当于是`一个连接一个线程`。

NIO：同步非阻塞IO，基于Reactor模型，客户端和channel进行通信，channel可以进行读写操作，通过多路复用器selector来轮询注册在其上的channel，而后再进行IO操作。这样的话，在进行IO操作的时候再用一个线程去处理就可以了，也就是`一个请求一个线程`。

> Reactor模型是什么?
>
> 1. 基于池化思想，避免为每个连接创建线程，连接完成后将业务处理交给线程池处理
> 2. 基于IO复用模型，多个连接共用同一个阻塞对象，不用等待所有的连接。遍历到有新数据可以处理时，操作系统会通知程序，线程跳出阻塞状态，进行业务逻辑处理
>
> .
>
> 简单来说：Reactor线程模型的思想就是基于IO复用和线程池的结合。

AIO：（一般都会把AIO和NIO、BIO放一块比较，这里简单提一下。）异步非阻塞IO，相比NIO更进一步，完全由操作系统来完成请求的处理，然后通知服务端开启线程去进行处理，因此是`一个有效请求一个线程`。

那么怎么理解同步和阻塞?

首先，可以认为一个IO操作包含两个部分：

1. 发起IO请求
2. 实际的IO读写操作

同步和异步在于第二个，实际的IO读写操作，如果操作系统帮你完成了再通知你，那就是异步，否则都叫做同步。

阻塞和非阻塞在于第一个，发起IO请求，对于NIO来说通过channel发起IO操作请求后，其实就返回了，所以是非阻塞。

NIO和BIO是非常重要的计算机知识，学习后会对整个计算机的理解更近一步，一次学会终身受益。JavaPub会单独写一篇深入图解NIO和BIO。

网上看到一个例子（一定要看，会对你有所帮助）：

> 一辆从 A 开往 B 的公共汽车上，路上有很多点可能会有人下车。司机不知道哪些点会有哪些人会下车，对于需要下车的人，如何处理更好？
>
> 1. 司机过程中定时询问每个乘客是否到达目的地，若有人说到了，那么司机停车，乘客下车。（类似阻塞式）
>
> 2. 每个人告诉售票员自己的目的地，然后睡觉，司机只和售票员交互，到了某个点由售票员通知乘客下车。（类似非阻塞）
>
> 很显然，每个人要到达某个目的地可以认为是一个线程，司机可以认为是 CPU 。在阻塞式里面，每个线程需要不断的轮询，上下文切换，以达到找到目的地的结果。而在非阻塞方式里，每个乘客（线程）都在睡觉（休眠），只在真正外部环境准备好了才唤醒，这样的唤醒肯定不会阻塞。

建议阅读：
https://www.cnblogs.com/aspirant/p/6877350.html
https://www.cnblogs.com/shoshana-kong/p/11228555.html

---

6. String、StringBuffer、StringBuilder 的区别是什么?

String是Immutable类的典型实现，被声明为 final class，除了hash这个属性其它属性都声明为final。它的不可变性，所以例如拼接字符串时候会产生很多无用的中间对象，如果频繁的进行这样的操作对性能有所影响。

StringBuffer、StringBuilder就是解决String的这个性能问题。

StringBuffer 是线程安全的，本质是一个线程安全的可修改的字符序列，把所有修改数据的方法都加上synchronized。

StringBuffer 线程不安全，但是性能更好。

---

7. 泛型是什么，有什么特点

泛型在编码中有非常广泛的使用（jdk5引入），你一定经常能见到类似这种写法 `<T>` 。

泛型提供了编译时类型安全检测机制，允许在编译时检测到非法的类型。**本质**是参数化类型。

1. 把类型当作是参数一样传递
2. <数据类型>只能是引用类型

泛型：就是一种不确定的数据类型。

泛型的好处：

1. 省略了强转的代码。
2. 可以把运行时的问题提前到编译时期。

> 引入泛型主要想实现一个通用的、可以处理不同类型的方法

泛型擦除:

泛型时提供给javac编译器使用的，用于限定集合的输入类型，让编译器在源代码级别上，避免向集合中插入非法数据。但编译器编译完带有泛型的java程序后，生成的class文件中不再带有泛型信息，以此使程序运行效率不受影响，这个过程称为擦除。

JVM并不知道泛型的存在，因为泛型在编译阶段就已经被处理成普通的类和方法； 处理机制是通过类型擦除，擦除规则：

- 若泛型类型没有指定具体类型，用Object作为原始类型；
- 若有限定类型< T exnteds XClass >，使用XClass作为原始类型；
- 若有多个限定< T exnteds XClass1 & XClass2 >，使用第一个边界类型XClass1作为原始类型；

---

8. final 有哪些用法

final关键字有四个常见用法。

final修饰一个类

当 final 关键字用来修饰一个类的时候，表明这个类不能有任何的子类，也就是说这个类不能被继承。

final类中的所有成员方法都会被隐式地指定为final方法，也就是说一个类如果是final的，那么其中所有的成员方法都无法进行覆盖重写。

```
public final class 类名称 {

    // ...

 }
```

final修饰一个方法

当 final 关键字用来修饰一个方法的时候，这个方法就是最终方法，也就是不能被覆盖重写。

```
修饰符 final 返回值类型 方法名称(参数列表) {

    // 方法体

}
```

注意：对于类、方法来说，abstract 关键字和 final 关键字不能同时使用，因为矛盾。

final修饰一个局部变量

一旦使用 final 用来修饰局部变量，那么这个变量就不能进行更改「一次赋值，终生不变」。

1. 对于基本类型来说，不可变说的是变量当中的数据不可改变；
2. 对于引用类型来说，不可变说的是变量当中的地址值不可改变。

final修饰一个成员变量

对于成员变量来说，如果使用 final 关键字修饰，那么这个变量也照样是不可变。

1. 由于成员变量具有默认值，所以用了 final 之后**必须**手动赋值，不会再给默认值了；
2. 对于 final 的成员变量，要么使用直接赋值，要么通过构造方法赋值，**必须**二者选其一；
3. **必须**保证类当中所有重载的构造方法都最终会对 final 的成员变量进行赋值。

---

9. 说一下Java注解

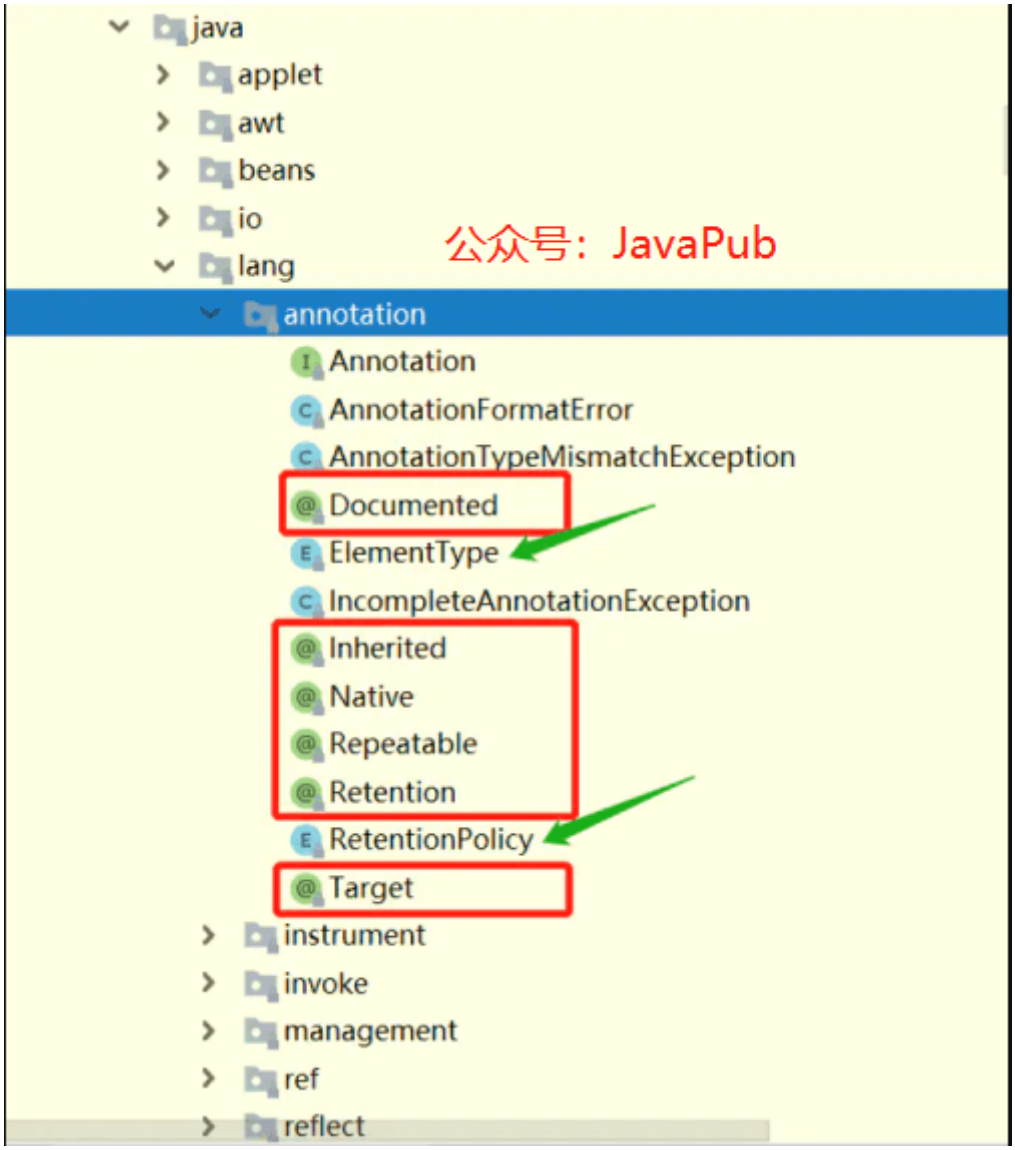
在Java编程中，注解非常常见，注解的本质是什么?

注解大致分为以下三种：

1. Java原生注解 如@Override，@Deprecated 等。大多用于 [标记] 和 [检查] 。
2. 第三方注解，如 Spring、Mybatis等定义的注解（@Controller，@Data）。
3. 自定义注解。

Java原生除了提供基本注解，还提供了 meta-annotation（元注解）。这些类型和它们所支持的类在java.lang.annotation包中可以找到。

1. @Target
2. @Retention
3. @Documented
4. @Inherited



一般比较常用的有 @Target，@Retention。`@Target`表示这个注解可以修饰那些地方（比如类、方法、成员变量），`@Retention`主要是设置注解的生命周期。

> 这是你一定会被问，
>
> 1. 有使用过注解吗?
> 2. 你是怎么使用的?

注解有一个非常常见的使用场景，大家可以用这个来理解学习。

场景一：自定义注解+拦截器 实现登录校验

实现功能：

接下来，我们使用springboot拦截器实现这样一个功能，如果方法上加了@LoginRequired，则提示用户该接口需要登录才能访问，否则不需要登录。

首先定义一个LoginRequired注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
    public @interface LoginRequired {

}
```

然后写两个简单的接口，访问sourceA，sourceB资源

```
@RestController
public class IndexController {

    @GetMapping("/sourceA")
    public String sourceA(){
        return "你正在访问sourceA资源";
    }

    @GetMapping("/sourceB")
    public String sourceB(){
        return "你正在访问sourceB资源";
    }

}
```

很简单的俩个接口，没添加拦截器之前成功访问

实现 spring 的 HandlerInterceptor 类先实现拦截器，但不拦截，只是简单打印日志，如下：

```
public class SourceAccessInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("进入拦截器了");
        return true;
    }

    @Override
```

```
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {

    }
}
```

实现spring类 WebMvcConfigurer，创建配置类把拦截器添加到拦截器链中

```
@Configuration
public class InterceptorTrainConfigurer implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
SourceAccessInterceptor()).addPathPatterns("/**");
    }
}
```

拦截成功如下

```
2019-07-04 08:09:47.382  INFO 14188 --- [nio-8080-e
2019-07-04 08:09:47.382  INFO 14188 --- [nio-8080-e
2019-07-04 08:09:47.387  INFO 14188 --- [nio-8080-e
进入拦截器了
```

在 sourceB 方法上添加我们的登录注解 @LoginRequired

```
@RestController
public class IndexController {

    @GetMapping("/sourceA")
    public String sourceA(){
        return "你正在访问sourceA资源";
    }

    @LoginRequired
    @GetMapping("/sourceB")
    public String sourceB(){
        return "你正在访问sourceB资源";
    }

}
```

简单实现登录拦截逻辑

```
    @Override
```

```
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("进入拦截器了");

        // 反射获取方法上的LoginRequred注解
        HandlerMethod handlerMethod = (HandlerMethod)handler;
        LoginRequired loginRequired =
handlerMethod.getMethod().getAnnotation(LoginRequired.class);
        if(loginRequired == null){
            return true;
        }

        // 有LoginRequired注解说明需要登录，提示用户登录
        response.setContentType("application/json; charset=utf-8");
        response.getWriter().print("你访问的资源需要登录");
        return false;
    }
```

运行成功，访问sourceB时需要登录了，访问sourceA则不用登录。

sourceA

sourceB

场景二：自定义注解+AOP 实现日志打印

先导入切面需要的依赖包

```
<dependency>
      <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

定义一个注解@MyLog

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyLog {

}
```

定义一个切面类，见如下代码注释理解：

```
@Aspect // 1.表明这是一个切面类
@Component
public class MyLogAspect {

    // 2. PointCut表示这是一个切点，@annotation表示这个切点切到一个注解上，后面带该注解的全类名
    // 切面最主要的就是切点，所有的故事都围绕切点发生
    // logPointCut()代表切点名称
    @Pointcut("@annotation(com.javapub.blog.MyLog)")
    public void logPointCut(){};

    // 3. 环绕通知
    @Around("logPointCut()")
    public void logAround(ProceedingJoinPoint joinPoint){
        // 获取方法名称
        String methodName = joinPoint.getSignature().getName();
        // 获取入参
        Object[] param = joinPoint.getArgs();

        StringBuilder sb = new StringBuilder();
        for(Object o : param){
            sb.append(o + "; ");
        }
        System.out.println("进入[" + methodName + "]方法,参数为:" + sb.toString());

        // 继续执行方法
        try {
            joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println(methodName + "方法执行结束");

    }
}
```

在步骤二中的IndexController写一个sourceC进行测试，加上我们的自定义注解：

```
    @MyLog
    @GetMapping("/sourceC/{source_name}")
    public String sourceC(@PathVariable("source_name") String sourceName){
        return "你正在访问sourceC资源";
    }
```

启动springboot web项目，输入访问地址


```
进入拦截器了
进入[sourceC]方法,参数为:zebinSource;
sourceC方法执行结束
```

切面成功

有些面试官喜欢问，注解三要素是哪些：

1. 注解声明、
2. 使用注解的元素、
3. 操作注解使其起作用(注解处理器)

---

10. Java创建对象有几种方式

Java中有5种创建对象的方式，下面给出它们的例子

使用new关键字	} → 调用了构造函数
使用Class类的newInstance方法	} → 调用了构造函数
使用Constructor类的newInstance方法	} → 调用了构造函数
使用clone方法	} → 没有调用构造函数
使用反序列化	} → 没有调用构造函数

使用new关键字

```
User user = new User();
```

使用Class类的newInstance方法

我们也可以使用Class类的newInstance方法创建对象。这个newInstance方法调用无参的构造函数创建对象。

```
Employee emp = (Employee)
Class.forName("org.javapub.blog.Employee").newInstance();
或者
Employee emp2 = Employee.class.newInstance();
```

使用Constructor类的newInstance方法

和Class类的newInstance方法很像， java.lang.reflect.Constructor类里也有一个newInstance方法可以创建对象。我们可以通过这个newInstance方法调用有参数的和私有的构造函数。

```
Constructor<Employee> constructor = Employee.class.getConstructor();
Employee emp3 = constructor.newInstance();
```

使用clone方法

无论何时我们调用一个对象的clone方法，jvm就会创建一个新的对象，将前面对象的内容全部拷贝进去。用clone方法创建对象并不会调用任何构造函数。

要使用clone方法，我们需要先实现Cloneable接口并实现其定义的clone方法。

```
Employee emp4 = (Employee) emp3.clone();
```

使用反序列化

当我们序列化和反序列化一个对象，jvm会给我们创建一个单独的对象。在反序列化时，jvm创建对象并不会调用任何构造函数。

为了反序列化一个对象，我们需要让我们的类实现Serializable接口

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));
Employee emp5 = (Employee) in.readObject();
```

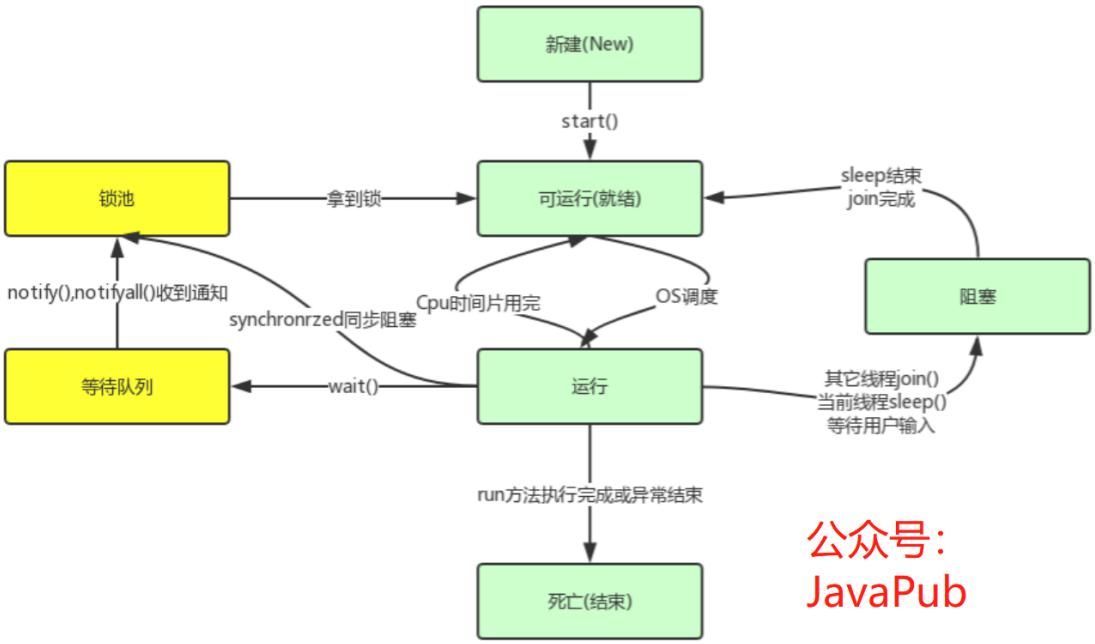
Java并发

1. start()方法和run()方法的区别

如果只是调用 run() 方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

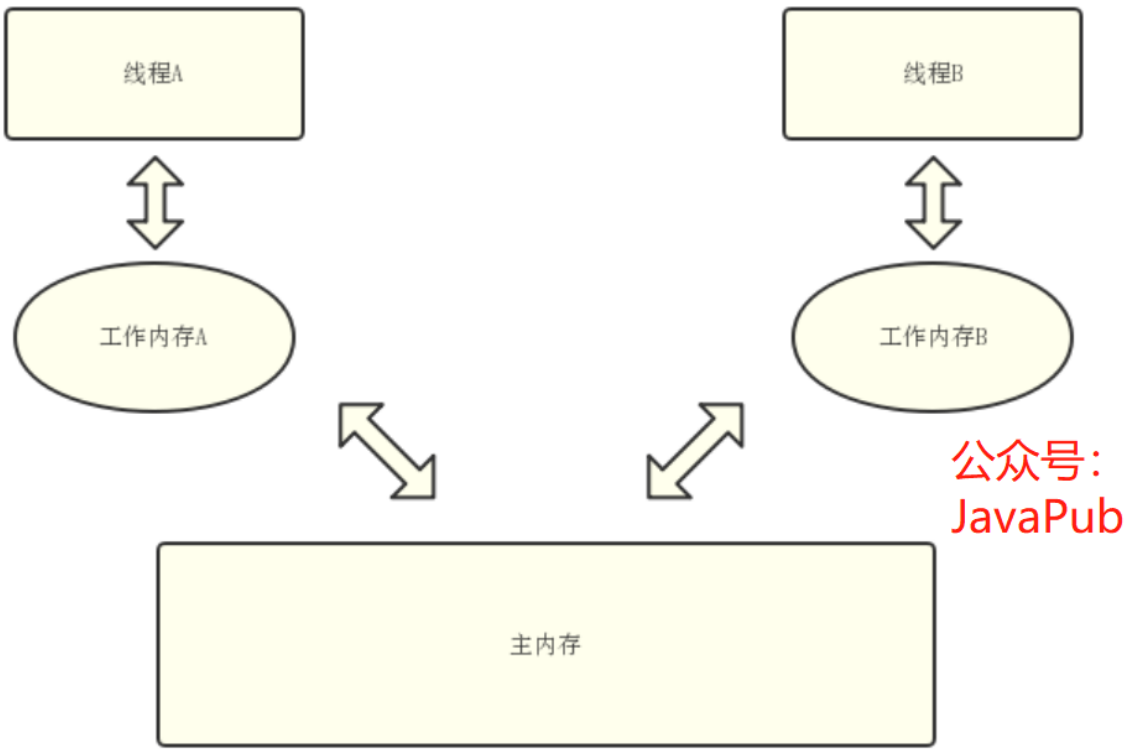
只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。

参考图：



2. volatile关键字的作用

volatile `英 ['vɒlətaɪl]`，第一个想到的一定是**保证内存可见性**（Memory Visibility）。可见性是性对于线程而言。



上图是Java内存模型，所有线程的共享变量都放在主内存中，每一个线程都有一个独有的工作内存，每个线程不直接操作在主内存中的变量，而是将主内存上变量的副本放进自己的工作内存中，只操作工作内存中的数据。当修改完毕后，再把修改后的结果放回到主内存中。每个线程都只操作自己工作内存中的变量，无法直接访问对方工作内存中的变量，线程间变量值的传递需要通过主内存来完成。

很明显，在并发环境下一定会发生脏数据问题。

使用volatile变量能够保证:

1. 每次读取前必须先从主内存刷新最新的值。
2. 每次写入后必须立即同步回主内存当中。

> 也就是说，volatile关键字修饰的变量看到的随时是自己的最新值。

防止指令重排

在基于偏序关系的Happens-Before内存模型中，指令重排技术大大提高了程序执行效率。但是也引入一个新问题：被部分初始化的对象

例子:

```
创建一个对象
instance = new Singleton();
```

它并不是一个原子操作。事实上，它可以"抽象"为下面几条JVM指令:

```
memory = allocate();    //1：分配对象的内存空间
initInstance(memory);   //2：初始化对象
instance = memory;      //3：设置instance指向刚分配的内存地址
```

上面操作2依赖于操作1，但是操作3并不依赖于操作2，所以JVM可以以"优化"为目的对它们进行重排序，经过重排序后如下:

```
memory = allocate();    //1：分配对象的内存空间
instance = memory;      //3：设置instance指向刚分配的内存地址（此时对象还未初始化）
initInstance(memory);   //2：初始化对象
```

可以看到指令重排之后，操作 3 排在了操作 2 之前，即引用instance指向内存memory时，这段崭新的内存还没有初始化。由于instance已经指向了一块内存空间，从而返回 instance!=null，用户得到了没有完成初始化的"半个"单例。

但是有一点：volatile不保证原子性。

> 这里有一篇生产环境使用volatile的例子：https://mp.weixin.qq.com/s/s1cwut9WvUSrMYw_6UK3sg

---

3. sleep方法和wait方法有什么区别

要了解sleep和wait，首先需要了解Java线程的6种状态。

```
#下面是Java线程的6种状态
1. 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。
2. 运行(RUNNABLE)：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为"运行"。
线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。
3. 阻塞(BLOCKED)：表示线程阻塞于锁。
4. 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
5. 超时等待(TIMED_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回。
6. 终止(TERMINATED)：表示该线程已经执行完毕。
```

sleep 休眠方法

```
static void sleep(long ms)
```

该方法会使当前线程进入阻塞状态指定毫秒，当阻塞指定毫秒后，当前线程会重新进入Runnable状态，等待划分时间片。

sleep方法属于Thread类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了sleep方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。

wait 方法一般是跟notify方法连用的

多线程之间需要协调工作。如果条件不满足则等待。当条件满足时，等待该条件的线程将被唤醒。在Java中，这个机制实现依赖于wait/notify或wait/notifyAll。

object.wait()让当前线程进入不可运行状态，如sleep()一样，但不同的是wait方法从一个对象调用，而不是从一个线程调用；我们称这个对象为"锁定对象（lockObj）"。在lockObj.wait()被调用之前，当前线程必须在lockObj上同步（synchronize）；然后调用wait()后释放这个锁，并将线程增加到与lockObj相关的"等待名单（wait list）"。然后，另一个在同一个lockObj锁定（synchronize）的方法可以调用lockObj.nofity()。这会唤醒原来等待的线程。基本上，wait() / notify()就像sleep() / interrupt()，只是活动线程不需要直接指向一个睡眠线程，他们只需要共享锁对象（lockObj）。

到这里你是否明白这个问题，如果不明白来JavaPub，后续一篇代码分析，马上安排。

---

4. 如何停止一个正在运行的线程?

最直观的一定是 Thread.stop，但是它是不推荐的，并且已经废弃。看一下官方说明
https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

```
*       This method is inherently unsafe.   Stopping a thread with
*       Thread. stop causes it to unlock all of the monitors that it
*       has locked (as a natural consequence of the unchecked
*       <code>ThreadDeath</code> exception propagating up the stack).   If
*       any of the objects previously protected by these monitors were in
*       an inconsistent state, the damaged objects become visible to
*       other threads, potentially resulting in arbitrary behavior.   Many
*       uses of <code>stop</code> should be replaced by code that simply
*       modifies some variable to indicate that the target thread should
*       stop running.   The target thread should check this variable
*       regularly, and return from its run method in an orderly fashion
*       if the variable indicates that it is to stop running.   If the
*       target thread waits for long periods (on a condition variable,
*       for example), the <code>interrupt</code> method should be used to
*       interrupt the wait.
```

小结：
简单来说，Thread.stop()不安全，已不再建议使用。

方法一：

使用 interrupt 方法中断线程。

interrupt()方法的使用效果并不像for+break语句那样，马上就停止循环。调用interrupt方法是在当前线程中打了一个停止标志，并不是真的停止线程。

需要 this.isInterrupted(): 测试线程是否真的已经中断。

方法二：

最好的一种方法，使用标志位停止。

run() 方法中做标识符，保证优雅的停止服务。

---

5. java如何实现多线程之间的通讯和协作？(如何在两个线程间共享数据？)

volatile关键字方式

volatile有两大特性，一是可见性，二是有序性，禁止指令重排序，其中可见性就是可以让线程之间进行通信。

等待/通知机制

等待通知机制是基于wait和notify方法来实现的，在一个线程内调用该线程锁对象的wait方法，线程将进入等待队列进行等待直到被通知或者被唤醒。

也就是通过**等待/通知机制** 让多个线程协作

join方式

join其实合理理解成是线程合并，当在一个线程调用另一个线程的join方法时，当前线程阻塞等待被调用join方法的线程执行完毕才能继续执行，所以join的好处能够保证线程的执行顺序，但是如果调用线程的join方法其实已经失去了并行的意义，虽然存在多个线程，但是本质上还是串行的，最后join的实现其实是基于等待通知机制的。

threadLocal方式

threadLocal方式的线程通信，不像以上三种方式是多个线程之间的通信，它更像是一个线程内部的通信，将当前线程和一个map绑定，在当前线程内可以任意存取数据，减省了方法调用间参数的传递。

---

6. 什么是ThreadLocal?

定义：线程局部变量是局限于线程内的变量，属于线程自身所有，不在多个线程间共享。java提供ThreadLocal类 来支持线程局部变量，是一个实现线程安全的方式。

作用：ThreadLocal 是一种以空间换时间的做法，在每一个 Thread 里面维护了一个 ThreadLocal.ThreadLocalMap 把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了。

---

7. Java 中 CountDownLatch 和 CyclicBarrier 有什么不同?

概念：

CountDownLatch 是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。简单来说：CountDownLatch 是一个计数器，可以保证线程之间的顺序执行把线程从并发状态调整为串行状态保证了线程的执行顺序。(只可以使用一次)

CyclicBarrier 是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。典型场景：可以用于多线程计算数据，最后合并计算结果。（可以多次使用）

分享一个直观的代码：

```java
package com.javapub.test;

import java.util.concurrent.CountDownLatch;

/**
 * @Author: JavaPub
 * @License: https://github.com/Rodert/
 * @Contact: https://javapub.blog.csdn.net/
 * @Date: 2022/1/1 16:50
 * @Version: 1.0
 * @Description: countDownLatch 可以保证线程之间的顺序执行把线程从并发状态调整为串行状态保证了线程的执行顺序。
 * demo效果：当打印完B，再打印C。
 */

class ThreadA extends Thread {

    private CountDownLatch down;

    public ThreadA(CountDownLatch down) {
        this.down = down;
    }

    @Override
    public void run() {
        System.out.println("A");
        try {
            down.await();//相当于wait(),调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("C");
    }
}

class ThreadB extends Thread {
    private CountDownLatch down;

    public ThreadB(CountDownLatch down) {
        this.down = down;
    }

    @Override
    public void run() {
        System.out.println("B");
        System.out.println(down.getCount());
        down.countDown();//将count值减1
    }
}

public class Test {
    public static void main(String[] args) {
        CountDownLatch down = new CountDownLatch(1);//创建1个计数器
        new ThreadA(down).start();
        new ThreadB(down).start();
    }
```

```
}

/*输出
A
B
C
 */
```

```
package com.roundyuan.fanggateway.test;

import java.util.concurrent.CyclicBarrier;

/**
 * @Author: JavaPub
 * @License: https://github.com/Rodert/
 * @Contact: https://javapub.blog.csdn.net/
 * @Date: 2022/1/2 13:42
 * @Version: 1.0
 * @Description: CyclicBarrier
 */

public class CyclicBarrierDemo {

    static class TaskThread extends Thread {

        CyclicBarrier barrier;

        public TaskThread(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                System.out.println(getName() + " 到达栅栏 A");
                barrier.await();
                System.out.println(getName() + " 冲破栅栏 A");

                Thread.sleep(2000);
                System.out.println(getName() + " 到达栅栏 B");
                barrier.await();
                System.out.println(getName() + " 冲破栅栏 B");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        int threadNum = 5;
        CyclicBarrier barrier = new CyclicBarrier(threadNum, new Runnable() {

            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " 完成最后任
务");
```

```
            }
        });

        for (int i = 0; i < threadNum; i++) {
            new TaskThread(barrier).start();
        }
    }
}

/*
Thread-3 到达栅栏 A
Thread-1 到达栅栏 A
Thread-4 到达栅栏 A
Thread-2 到达栅栏 A
Thread-0 到达栅栏 A
Thread-2 完成最后任务
Thread-2 冲破栅栏 A
Thread-0 冲破栅栏 A
Thread-4 冲破栅栏 A
Thread-3 冲破栅栏 A
Thread-1 冲破栅栏 A
Thread-4 到达栅栏 B
Thread-0 到达栅栏 B
Thread-2 到达栅栏 B
Thread-1 到达栅栏 B
Thread-3 到达栅栏 B
Thread-3 完成最后任务
Thread-3 冲破栅栏 B
Thread-0 冲破栅栏 B
Thread-4 冲破栅栏 B
Thread-1 冲破栅栏 B
Thread-2 冲破栅栏 B
 */
```

网上看到一个比较形象一个例子：

CountDownLatch：
宿管阿姨，晚上关宿舍大门睡觉，需要等到所有学生回寝，才能关门睡觉，学生之间不用相互等待，回寝就能睡觉。 （学生就是各个线程，宿管阿姨就是监听CountDownLatch为0后要执行的。）

CyclicBarrier：
家庭聚餐，等待家庭成员到齐才能开饭，家庭成员之间需要相互等待，直到最后一个到达，才能同时开饭。

---

8. 如何避免死锁?

从上图我们就可以看出，产生死锁就是俩个或多个线程在申请资源时，自己需要的资源别别人持有、并阻塞。所以导致死锁。

如何解决：

1. 减小锁的范围，尽量保证之锁定自己需要的资源，减小交叉持有资源情况
2. 但是有些时候不得不持有多个资源，比如**银行转账**，我们必须同时获得两个账户上的锁，才能进行操作，两个锁的申请必须发生交叉。这时我们也可以打破死锁的那个闭环，在涉及到要同时申请两个锁的方法中，总是以相同的顺序来申请锁，比如总是先申请 id 大的账户上的锁 ，然后再申请 id 小的账户上的锁，这样就无法形成导致死锁的那个闭环。
3. 我们知道导致死锁有一个因素是阻塞，所以如果我们不使用默认阻塞的锁，也是可以避免死锁的。我们可以使用 ReentrantLock.tryLock() 方法，在一个循环中，如果 tryLock() 返回失败，那么就释放以及获得的锁，并睡眠一小段时间。这样就打破了死锁的闭环。

```
package com.roundyuan.fanggateway.test;

import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @Author: JavaPub
 * @License: https://github.com/Rodert/
 * @Contact: https://javapub.blog.csdn.net/
 * @Date: 2022/1/2 14:38
 * @Version: 1.0
 * @Description: ReentrantLock
 */

public class DeadLock {

    private static Lock lock1 = new ReentrantLock();
    private static Lock lock2 = new ReentrantLock();

    public static void deathLock() {
        new Thread() {
            @Override
            public void run() {
                while (true) {
                    if (lock1.tryLock()) {
                        try {
                            //如果获取成功则执行业务逻辑，如果获取失败，则释放lock1的锁，
自旋重新尝试获得锁
                            if (lock2.tryLock()) {
```

```
                                try {
                                    System.out.println("Thread1：已成功获取 lock1
and lock2 ...");
                                    break;
                                } finally {
                                    lock2.unlock();
                                }
                            }
                        } finally {
                            lock1.unlock();
                        }
                    }
                    System.out.println("Thread1：获取锁失败，重新获取---");
                    try {
                        //防止发生活锁
                        TimeUnit.NANOSECONDS.sleep(new Random().nextInt(100));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();

        new Thread() {
            @Override
            public void run() {
                while (true) {
                    if (lock2.tryLock()) {
                        try {
                            //如果获取成功则执行业务逻辑，如果获取失败，则释放lock2的锁，
自旋重新尝试获得锁
                            if (lock1.tryLock()) {
                                try {
                                    System.out.println("Thread2：已成功获取 lock2
and lock1 ...");
                                    break;
                                } finally {
                                    lock1.unlock();
                                }
                            }
                        } finally {
                            lock2.unlock();
                        }
                    }
                    System.out.println("Thread2：获取锁失败，重新获取---");
                    try {
                        //防止发生活锁
                        TimeUnit.NANOSECONDS.sleep(new Random().nextInt(100));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 5; i++) {
```

```
            deathLock();
        }
    }
}
```

说起死锁，银行家算法非常有必要了解：

> 银行家算法（Banker's Algorithm）是一个避免死锁（Deadlock）的著名算法，是由艾兹格·迪杰斯特拉在1965年为T.H.E系统设计的一种避免死锁产生的算法。它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。
> 1、操作系统按照银行家指定的规则为进程分配资源，当进程首次申请资源时，需要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请资源分配资源，否则就推迟分配；
> 2、当进程在执行中继续申请资源时，先测试该进程本次申请的资源数，是否超过了该资源剩余的总量，若超过则拒绝分配资源，若能满足则按当前的申请量分配资源，否则也要推迟分配。

参考阅读：

如何快速排查死锁？如何避免死锁？
https://zhuanlan.zhihu.com/p/136294283

---

9. Java 中 synchronized 和 ReentrantLock 有什么不同?

等待可中断:

使用synchronized，不能被中断。synchronized 也可以说是Java提供的原子性内置锁机制。内部锁扮演了互斥锁（mutual exclusion lock ，mutex）的角色，一个线程引用锁的时候，别的线程阻塞等待。

使用ReentrantLock。等待了很长时间以后，可以中断等待，转而去做别的事情。

公平锁:

公平锁是指多个线程在等待同一个锁时，必须按照申请的时间顺序来依次获得锁；而非公平锁则不能保证这一点。非公平锁在锁被释放时，任何一个等待锁的线程都有机会获得锁。 synchronized的锁是非公平锁，ReentrantLock默认情况下也是非公平锁，但可以通过带布尔值的构造函数要求使用公平锁。

还有大家已知的俩点：

1. synchronized是独占锁，加锁和解锁的过程自动进行，易于操作，但不够灵活。ReentrantLock也是独占锁，加锁和解锁的过程需要手动进行，不易操作，但非常灵活。
2. synchronized可重入，因为加锁和解锁自动进行，不必担心最后是否释放锁；ReentrantLock也可重入，但加锁和解锁需要手动进行，且次数需一样，否则其他线程无法获得锁。

---

10. 有三个线程 T1，T2，T3，怎么确保它们按顺序执行?

方法1:

线程内部顺序调用，T1、T2、T3。这个可能不是要考察的点，但也是一个方案。

方法2:

join()方法用于将线程由 "并行"变成"串行"，它用于等待其他线程的终止，在当前线程掉用了join()方法，那么当前线程将进入阻塞状态，等到另一个线程结束，当前线程再由阻塞状态转变成就绪状态，等待CPU的使用权。

```
package com.javapub.test;
```

```java
/**
 * @Author: JavaPub
 * @License: https://github.com/Rodert/
 * @Contact: https://javapub.blog.csdn.net/
 * @Date: 2022/1/2 15:20
 * @Version: 1.0
 * @Description:
 */


public class Test1 {

    public static void main(String[] args) {
        ThreadA threadA = new ThreadA();
        ThreadB threadB = new ThreadB(threadA);
        ThreadC threadC = new ThreadC(threadB);

        threadA.start();
        threadB.start();
        threadC.start();

    }

}

class ThreadA extends Thread {
    @Override
    public void run() {
        System.out.println("线程A");
    }
}

class ThreadB extends Thread {
    Thread threadA;

    public ThreadB() {
        // dosomething Auto-generated constructor stub
    }

    public ThreadB(Thread threadA) {
        this.threadA = threadA;
    }

    @Override
    public void run() {
        try {
            threadA.join();
        } catch (InterruptedException e) {
            // dosomething Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("线程B");
    }
}

class ThreadC extends Thread {
    Thread threadB;
```

```
    public ThreadC(Thread threadB) {
        this.threadB = threadB;
    }

    @Override
    public void run() {
        try {
            threadB.join();
        } catch (InterruptedException e) {
            // dosomething Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("线程C");
    }
}
```

信号量方式 java.util.concurrent.Semaphore 英 ['seməfɔ:(r)]
待研究

Java 容器

1. 请说一下Java容器集合的分类，各自的继承结构

Java 容器分为 Collection 和 Map 两大类，其下又有很多子类，如下所示：

Collection包括：List、ArrayList、LinkedList、Vector、Stack、Set、HashSet、LinkedHashSet、TreeSet

Map包括：HashMap、LinkedHashMap、TreeMap、ConcurrentHashMap、Hashtable

2. Collection 和 Collections 有什么区别?

Collection 是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法，所有集合都是它的子类，比如 List、Set 等。

Collections 是一个包装类，包含了很多静态方法，不能被实例化，就像一个工具类，比如提供的排序方法：Collections. sort(list)。

3. List、Set、Map 之间的区别是什么?

List、Set、Map 的区别主要体现在两个方面：元素是否有序、是否允许元素重复。

4. HashMap 和 Hashtable 有什么区别?

- HashMap 是非线程安全的，HashTable 是线程安全的。
- HashMap 的键和值都允许有 null 值存在，而 HashTable 则不行。
- 因为线程安全的问题，HashMap 效率比 HashTable 的要高。
- Hashtable 是同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而 Hashtable 适合于多线程环境。
- 一般现在 **不建议用 HashTable** ，
    - 一方面是因为 HashTable 是遗留类，内部实现很多没优化和冗余。

- 另外，即使在 **多线程** 环境下，现在也有同步的 **ConcurrentHashMap** 替代，没有必要因为是多线程而用 HashTable。

5. 说一下 HashMap 的实现原理?

HashMap 基于 Hash 算法实现的，我们通过 put(key,value)存储，get(key)来获取。当传入 key 时，HashMap 会根据
key. hashCode() 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。

当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash
冲突的个数比较少时，使用链表否则使用红黑树。

6. 谈谈 ArrayList 和 LinkedList 的区别

本质的区别来源于两者的底层实现：ArrayList的底层是数组，LinkedList的底层是双向链表。

数组拥有O(1)的查询效率，可以通过下标直接定位元素；链表在查询元素的时候只能通过遍历的方式查询，效率比数组低。

数组增删元素的效率比较低，通常要伴随拷贝数组的操作；链表增删元素的效率很高，只需要调整对应位置的指针即可。

以上是数组和链表的通俗对比，在日常的使用中，两者都能很好地在自己的适用场景发挥作用。

比如说我们常常用ArrayList代替数组，因为封装了许多易用的api，而且它内部实现了自动扩容机制，由于它内部维护了一个当前容量的指针size，直接往ArrayList中添加元素的时间复杂度是O(1)的，使用非常方便。

而LinkedList常常被用作Queue队列的实现类，由于底层是双向链表，能够轻松地提供先入先出的操作。

> 我觉得可以分两部分答，一个是数组与链表底层实现的不同，另一个是答ArrayList和LinkedList的实现细节。

7. 谈谈ArrayList和Vector的区别

两者的底层实现相似，关键的不同在于Vector的对外提供操作的方法都是用synchronized修饰的，也就是说Vector在并发环境下是线程安全的，而ArrayList在并发环境下可能会出现线程安全问题。

由于Vector的方法都是同步方法，执行起来会在同步上消耗一定的性能，所以在单线程环境下，Vector的性能是不如ArrayList的

除了线程安全这点本质区别外，还有一个实现上的小细节区别：ArrayList每次扩容的大小为原来的1.5倍；Vector可以指定扩容的大小，默认是原来大小的两倍。

> 可以顺带谈谈多线程环境下ArrayList的替代品，比如CopyOnWriteArrayList，但是要谈谈优缺点。

8. 请谈一谈 Java 集合中的 fail-fast 和 fail-safe 机制

fail-fast 是一种错误检测机制，Java 在适合单线程使用的集合容器中很好地实现了 fail-fast 机制，举一个简单的例子：在多线程并发环境下，A线程在通过迭代器遍历一个 ArrayList 集合，B线程同时对该集合进行增删元素操作，这个时候线程A就会抛出并发修改异常，中断正常执行的逻辑。

而fail-safe机制更像是一种对 fail-fast 机制的补充，它被广泛地实现在各种并发容器集合中。回头看上面的例子，如果线程A遍历的不是一个 ArrayList，而是一个 CopyOnWriteArrayList，则符合 fail-safe 机制，线程B可以同时对该集合的元素进行增删操作，线程A不会抛出任何异常。

要理解这两种机制的表象，我们得了解这两种机制背后的实现原理：

我们同样用 ArrayList 解释 fail-fast 背后的原理：首先 ArrayList 自身会维护一个 modCount 变量，每当进行增删元素等操作时，modCount 变量都会进行自增。当使用迭代器遍历 ArrayList 时，迭代器会新维护一个初始值等于 modCount 的 expectedModCount 变量，每次获取下一个元素的时候都会去检查 expectModCount 和 modCount 是否相等。在上面举的例子中，由于B线程增删元素会导致 modCount 自增，当A线程遍历元素时就会发现两个变量不等，从而抛出异常。

CopyOnWriteArrayList 所实现的 fail-safe 在上述情况下没有抛出异常，它的原理是：当使用迭代器遍历集合时，会基于原数组拷贝出一个新的数组（ArrayList的底层是数组），后续的遍历行为在新数组上进行。所以线程B同时进行增删操作不会影响到线程A的遍历行为。

9. HashMap是怎样确定key存放在数组的哪个位置的？JDK1.8

首先计算key的hash值，计算过程是：先得到key的hashCode（int类型，4字节），然后把hashCode的高16位与低16位进行异或，得到key的hash值。

接下来用key的hash值与数组长度减一的值进行按位与操作，得到key在数组中对应的下标。

9.1. 追问：为什么计算key的hash时要把hashCode的高16位与低16位进行异或？（变式：为什么不直接用key的hashCode）？

计算key在数组中的下标时，是通过hash值与数组长度减一的值进行按位与操作的。由于数组的长度通常不会超过2^16，所以hash值的高16位通常参与不了这个按位与操作。

为了让hashCode的高16位能够参与到按位与操作中，所以把hashCode的高16位与低16位进行异或操作，使得高16位的影响能够均匀稀释到低16位中，使得计算key位置的操作能够充分散列均匀。

10. 为什么要把链表转为红黑树，阈值为什么是8?

在极端情况下，比如说key的hashCode()返回的值不合理，或者多个密钥共享一个hashCode，很有可能会在同一个数组位置产生严重的哈希冲突。

这种情况下，如果我们仍然使用使用链表把多个冲突的元素串起来，这些元素的查询效率就会从O(1)下降为O(N)。为了能够在这种极端情况下仍保证较为高效的查询效率，HashMap选择把链表转换为红黑树，红黑树是一种常用的平衡二叉搜索树，添加，删除，查找元素等操作的时间复杂度均为O(logN)

至于阈值为什么是8，这是HashMap的作者根据概率论的知识得到的。当key的哈希码分布均匀时，数组同一个位置上的元素数量是成泊松分布的，同一个位置上出现8个元素的概率已经接近千分之一了，这侧面说明如果链表的长度达到了8，key的hashCode()肯定是出了大问题，这个时候需要红黑树来保证性能，所以选择8作为阈值。

追问：为什么红黑树转换回链表的阈值不是7而是6呢?

如果是7的话，那么链表和红黑树之间的切换范围值就太小了。如果我的链表长度不停地在7和8之间切换，那岂不是得来回变换形态？所以选择6是一种折中的考虑。

拓展题. 为什么 HashMap 数组的长度是2的幂次方?

因为这样能够提高根据 key 计算数组位置的效率。

HashMap 根据 key 计算数组位置的算法是：用 key 的 hash 值与数组长度减1的值进行按位与操作。

在我们正常人的思维中，获取数组的某个位置最直接的方法是对数组的长度取余数。但是如果被除数是2的幂次方，那么这个对数组长度取余的方法就等价于对数组长度减一的值进行按位与操作。

在计算机中，位运算的效率远高于取模运算，所以为了提高效率，把数组的长度设为2的幂次方。

> 所以一定要看一遍源码，相比于框架的源码，集合的源码简直太友好了。在笔试的时候可能还会考一些集合的使用，比如遍历，排序，比较等等，这些算是Java基础，用得多也就熟了。

JavaEE

1. JSP 有哪些内置对象？作用分别是什么?

JSP有9个内置对象：

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

2. 介绍一下 Servlet 生命周期

Servlet是运行在服务器端，以多线程的方式处理客户端请求的小程序。是sun公司提供的一套规范（规范的实现是接口）。

servlet的生命周期就是从servlet出现到消亡(销毁)的全过程。主要分为以下几个阶段：

加载类—>实例化(为对象分配空间)—>初始化(为对象的属性赋值)—>请求响应(服务阶段)—>销毁

详细介绍：

1. 加载

在下列时刻会加载Servlet（只执行一次）：

- 如果已经配置自动加载选项，则在启动服务器时自动加载 `web.xml` 文件中设置的 `<load-on-start>`；
- 服务器启动之后，客户机首次向Servlet发出请求时会加载；
- 重新加载Servlet时会进行一次加载；

2. 实例化

加载Servlet后，服务器创建一个Servlet实例。（只执行一次）

3. 初始化

- 调用 Servlet 的 `init()` 方法。在初始化阶段，Servlet 初始化参数被传递给 Servlet 配置对象 ServletConfig。（只执行一次）；

4. 请求处理

对于到达服务器的客户机请求，服务器创建针对此次请求的一个"请求对象"和一个"响应对象"。

服务器调用 Servlet 的 `service()` 方法，该方法用于传递"请求"和"响应"对象。

`service()` 方法从"请求"对象获得请求信息、处理该请求并用"响应"对象的方法将响应回传给客户端。

`service()` 方法可以调用其他方法来处理请求，例如 `doGet()、doPost()` 或其他方法。

5. 销毁

当服务器不需要 Servlet，或重新装入 Servlet 的新实例时，服务器会调用 Servlet 的 `destroy()` 方法。（只执行一次）；

3. Servlet和JSP的区别和联系

区别：

1. JSP是在HTML代码里面写Java代码；而Servlet是在Java代码中写HTML代码，Servlet本身是个Java类；
2. JSP使人们将显示和逻辑分隔称为可能，这意味着两者的开发可以并行进行；而Servlet并没有将两者分开；
3. Servlet独立地处理静态表示逻辑与动态业务逻辑，任何文件的变动都需要对此服务程序重新编译；JSP允许使用特殊标签直接嵌入到HTML页面，HTML内容与JAVA内容也可放在单独文件中，HTML内容的任何变动会自动编译装入到服务程序；
4. Servlet需要在web.xml中配置；而JSP无需配置；
5. 目前JSP主要用在视图层，负责显示；而Servlet主要用在控制层，负责调度；

联系：

1. 都是SUN公司推出的动态网页技术；
2. 先有Servlet，针对Servlet缺点推出JSP。JSP是Servlet的一种特殊形式，每个JSP页面就是一个Servlet实例，JSP页面由系统翻译成Servlet，Servlet再负责响应用户的请求。

4. JSP的执行过程

在JSP运行过程中，首先由客户端发出请求，Web服务器接收到请求之后，如果是第一次访问某个JSP页面，Web服务器对它进行一下三个操作：

1. 翻译

由.jsp变为.java，由JSP引擎完成。

2. 编译

由.java变为.class，由Java编译器实现。

3. 执行

由.class变为.html，用Java虚拟机执行编译文件，然后将执行结果返回给Web服务器，并最终返回给客户端。

如果不是第一次访问某个JSP页面，则只执行第三步，**所以第一次访问JSP较慢。**

5. Session和Cookie的区别和联系；说明在自己项目中如何使用?

Session 和 Cookie 都是会话(Seesion)跟踪技术。Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份。但是 Session 的实现依赖于 **Cookie，sessionId**(session的唯一标识需要存放在客户端).

1. cookie数据存放在客户的浏览器上，session数据放在服务器上。
2. cookie不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗，考虑到安全应当使用session。

3. session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用cookie。
4. 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。
5. 可以考虑将登录信息等重要信息存放为session，其他信息如果需要保留，可以放在cookie中。
6. 在程序开发过程中，我们可以在客户端每次与服务器交互时检查SessionID（Session中属性值，非HttpServlet环境开发中也可以用其它的Key值代替），用于会话管理。

- 将登陆信息等重要信息存放为SESSION
- 其他信息如果需要保留，可以放在COOKIE中，比如购物车

  购物车最好使用cookie，但是cookie是可以在客户端禁用的，这时候我们要使用cookie+数据库的方式实现，当从cookie中不能取出数据时，就从数据库获取。

6. 转发和重定向的联系和区别?

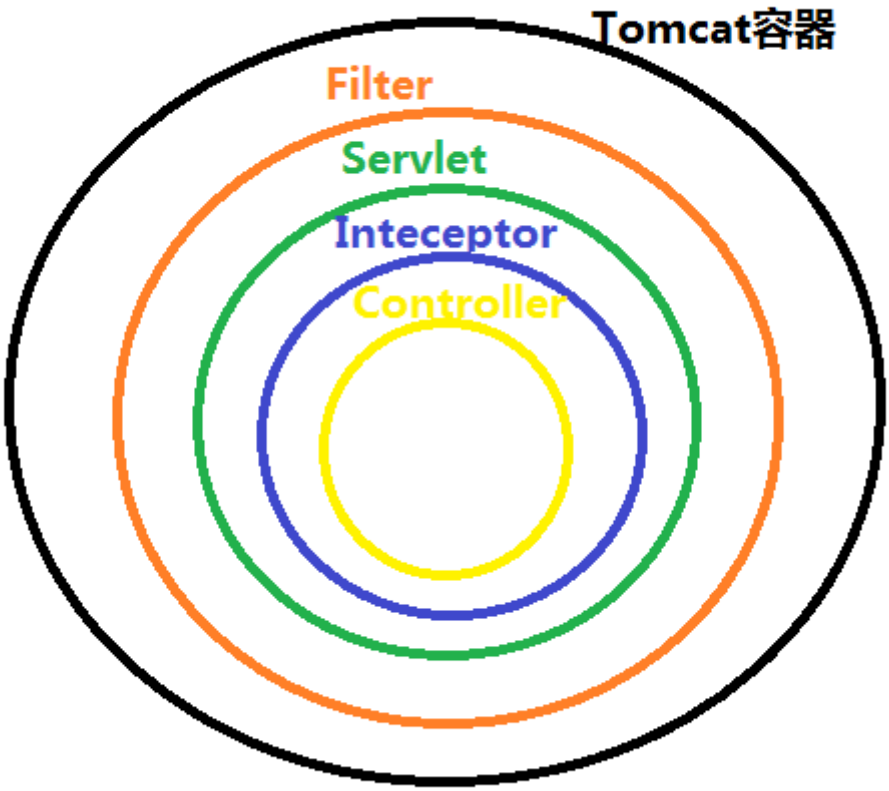
- **转发**：服务器端的跳转，路径不会发生改变（针对的是servlet），是服务器内部的处理，一次请求，请求对象不会变
- **重定向**：客户端的跳转，路径会发生改变，将要请求的路径和302重定向的状态码发给客户端浏览器，客户端浏览器将再次向服务器发出请求，不是同个请求，两次请求。

7. 拦截器和过滤器的区别

Spring 的拦截器与 Servlet 的 Filter 有相似之处，比如二者都是AOP编程思想的体现，都能实现权限检查、日志记录等。不同的是:

1. 使用范围不同: Filter 是 Servlet 规范规定的，只能用于Web程序中。而拦截器既可以用于Web程序，也可以用于 Application、Swing程序中。
2. 规范不同: Filter 是在 Servlet 规范中定义的，是 Servlet 容器支持的。而拦截器是在 Spring 容器内的，是 Spring 框架支持的。
3. 使用的资源不同:同其他的代码块一样，拦截器也是一个 Spring 的组件，归 Spring 管理，配置在 Spring 文件中，因此能使用 Spring 里的任何资源、对象，例如 Service 对象、数据源、事务管理等，通过 IoC 注入到拦截器即可;而Filter则不能。
4. 深度不同: Filter 在只在 Servlet 前后起作用。而拦截器能够深入到方法前后、异常抛出前后等，因此拦截器的使用具有更大的弹性。所以在 Spring 构架的程序中，要优先使用拦截器。

一张经典的图



8. 三次握手和四次挥手

> 这里是字面描述

三次握手:

1. 客户端向服务器发出连接请求等待服务器确认
2. 服务器向客户端返回一个响应告诉客户端收到了请求
3. 客户端向服务器再次发出确认信息,此时连接建立

四次挥手:

1. 客户端向服务器发出取消连接请求
2. 服务器向客户端返回一个响应,表示收到客户端取消请求
3. 服务器向客户端发出确认取消信息(向客户端表明可以取消连接了)
4. 客户端再次发送确认消息,此时连接取消

9. TCP和UDP的区别

1. TCP： 面向连接，UDP： 面向无连接
2. TCP： 传输效率低，UDP： 传输效率高(有大小限制，一次限定在64kb之内)
3. TCP：可靠，UDP： 不可靠

10. 如何解决跨域问题?

跨域指的是浏览器不能执行其它网站的脚本，它是由浏览器的**同源策略**造成的，是浏览器对 JavaScript 施加的安全限制。

所谓同源指的是：**协议、域名、端口号**都相同，只要有一个不相同，那么都是非同源。

解决方案:

1. 使用 ajax 的 jsonp。（这一点有些人是不知道的）
2. nginx 转发：利用 nginx 反向代理，将请求分发到部署相应项目的 tomcat 服务器，当然也不存在跨域问题。
3. 使用 CORS：写一个配置类实现 WebMvcConfigurer 接口或者配置 FilterRegistrationBean。

`CORS（Cross-Origin Resource Sharing）是一个W3C标准，全称“跨域资源共享”`

11. 什么是 CSRF 攻击？如何防御CSRF 攻击

CSRF（Cross-site request forgery） 跨站请求伪造。CSRF 攻击是在受害者毫不知情的情况下，以受害者名义伪造请求发送给受攻击站点，从而在受害者并未授权的情况下执行受害者权限下的各种操作。

CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

目前防御 CSRF 攻击主要有三种策略：

1. 验证 HTTP Referer 字段
2. 在请求地址中添加 token 并验证
3. 在 HTTP 头中自定义属性并验证

12. HTTP1.0和HTTP1.1和HTTP2.0的区别

1. **HTTP1.0**：无状态，无连接。
2. **HTTP1.1**：长连接，请求管道化，增加缓存处理，增加 Host 字段，支持断点传输。
3. **HTTP2.0**：二进制分帧，多路复用(连接共享)，头部压缩，服务器推送。

JVM

1. 说一说JVM的主要组成部分

点击放大看，一图胜千文

- 方法区和堆是所有线程共享的内存区域；而虚拟机栈、本地方法栈和程序计数器的运行是线程私有的内存区域，运行时数据区域就是我们常说的JVM的内存。
- 类加载子系统：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到运行时数据区中的方法区中。
- Java堆是Java虚拟机所管理的内存中最大的一块，也是垃圾回收的主要区域。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
- 方法区与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
- 程序计数器是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器，用来指示执行引擎下一条执行指令的地址。
- Java虚拟机栈也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、返回方法地址等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
- 本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。
- 执行引擎：根据程序计数器中存储的指令地址执行classes中的指令。
- 本地接口：与本地方法库交互，是其它编程语言交互的接口。

2. 说一下 JVM 的作用?

首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

3. 说一下堆栈的区别?

物理地址

堆的物理地址分配对象是不连续的。因此性能慢些。在GC的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）

栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

内存分别

堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS:

静态变量放在方法区
静态的对象还是放在堆。
程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

4. Java内存泄漏

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。

严格来说，只有对象不会再被程序用到了，但是GC又不能回收他们的情况，才叫内存泄漏。

理论上来说，Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收掉，自动从内存中清除。

但是，即使这样，Java也还是存在着内存泄漏的情况，java导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。

5. JVM 有哪些垃圾回收算法?

- 标记-清除算法：标记有用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。
- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半，消耗内存。

- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

6. 说一下 JVM 有哪些垃圾回收器?

7. 说一下类加载的执行过程?

- **加载**：根据查找路径找到相应的 class 文件然后装载入内存中；

- **验证**：检查加载的 class 文件的正确性；
- **准备**：给类中的静态变量分配内存空间；
- **解析**：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
- **初始化**：对静态变量和静态代码块执行初始化工作。

---

8. 什么是双亲委派模型？为什么要使用双亲委派模型?

什么是双亲委派模型

- 当需要加载一个类的时候，子类加载器并不会马上去加载，而是依次去请求父类加载器加载
- 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器；
- 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。

为什么要使用双亲委派模型

可以防止内存中出现多份同样的字节码，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，多个类加载器都去加载这个类到内存中，系统中将会出现多个不同的Object类，那么类之间的比较结果及类的唯一性将无法保证，而且如果不使用这种双亲委派模型将会给虚拟机的安全带来隐患。所以，要让类对象进行比较有意义，前提是他们要被同一个类加载器加载。

9. CMS垃圾清理的过程

CMS整个过程比之前的收集器要复杂，整个过程分为4个主要阶段，即初始标记阶段、并发标记阶段、重新标记阶段和并发清除阶段。(涉及STW的阶段主要是：初始标记 和 重新标记 stop-the-world)

- **初始标记**（Initial-Mark）阶段：在这个阶段中，程序中所有的工作线程都将会因为“stop-the-world”机制而出现短暂的暂停，这个阶段的主要任务仅仅只是标记出 GC Roots 能直接关联到的对象。一旦标记完成之后就会恢复之前被暂停的所有应用线程。由于直接关联对象比较小，所以这里的速度非常快。
- **并发标记**（Concurrent-Mark）阶段：从 Gc Roots 的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。
- **重新标记**（Remark）阶段：由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分

对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。
- **并发清除**（Concurrent-Sweep）阶段：此阶段清理删除掉标记阶段判断的已经死亡的对象，释放内存空间。由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的

10. 常用的 JVM 调优的参数都有哪些?

- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
- -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
- –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；
- -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；
- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
- -XX:+PrintGC：开启打印 gc 信息；
- -XX:+PrintGCDetails：打印 gc 详细信息。

Kafka

在面试kafka中，一定要了解为什么要用kafka、及kafka的架构等基本概念，才能对面试中的问题得心应手。

术语0. Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

ISR:In-Sync Replicas 副本同步队列

AR:Assigned Replicas 所有副本

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

术语0. Kafka中的HW、LEO、LSO、LW等分别代表什么?

HW:High Watermark 高水位，取一个partition对应的ISR中最小的LEO作为HW，consumer最多只能消费到HW所在的位置上一条信息。

LEO:LogEndOffset 当前日志文件中下一条待写信息的offset
HW/LEO这两个都是指最后一条的下一条的位置而不是指最后一条的位置。

LSO:Last Stable Offset 对未完成的事务而言，LSO 的值等于事务中第一条消息的位置(firstUnstableOffset)，对已完成的事务而言，它的值同 HW 相同

LW:Low Watermark 低水位, 代表 AR 集合中最小的 logStartOffset 值

1. kafka 是什么？有什么作用?

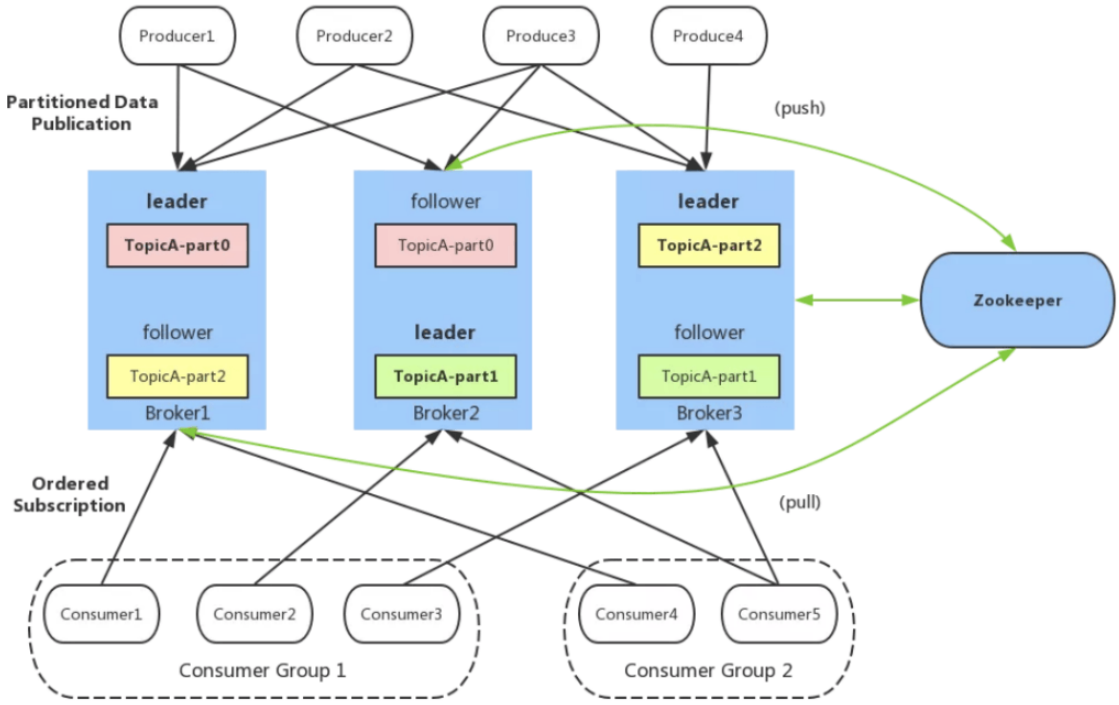
Kafka 是一个分布式的流式处理平台，它以高吞吐、可持久化、可水平扩展、支持流数据处理等多种特性而被广泛使用

主要功能体现于三点：

- **消息系统**：kafka与传统的消息中间件都具备**系统解耦、冗余存储、流量削峰、缓冲、异步通信、扩展性、可恢复性**等功能。与此同时，kafka还提供了大多数消息系统难以实现的消息顺序性保障及回溯性消费的功能。
- **存储系统**：kafka把**消息持久化到磁盘**，相比于其他基于内存存储的系统而言，有效的降低了消息丢失的风险。这得益于其消息持久化和多副本机制。也可以将kafka作为长期的存储系统来使用，只需要把对应的数据保留策略设置为“永久”或启用主题日志压缩功能。
- **流式处理平台**：kafka为流行的流式处理框架提供了可靠的数据来源，还提供了一个完整的流式处理框架，比如窗口、连接、变换和聚合等各类操作。

2. kafka 的架构是怎么样的?

> 这是一个基本概念的题目，一定要掌握。



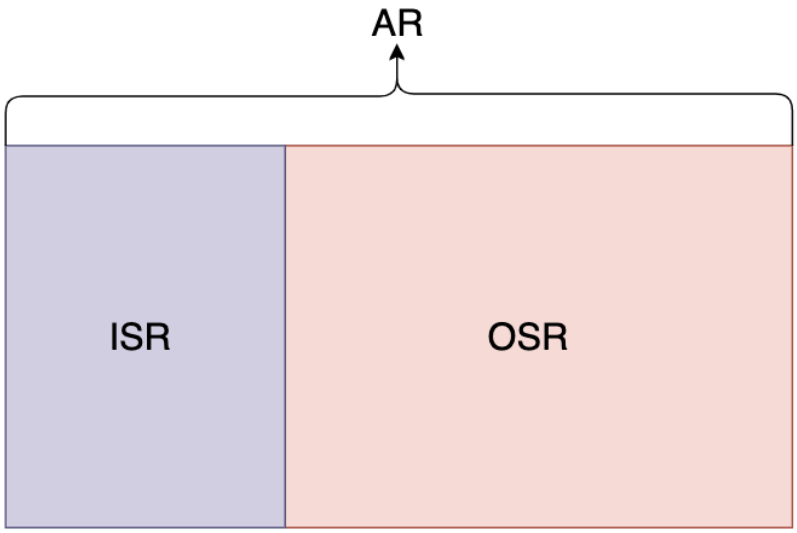
一个典型的 kafka 体系架构包括若干 Producer、若干 Consumer、以及一个 Zookeeper 集群（在2.8.0版本中移，除了 Zookeeper,通过 KRaft 进行自己的集群管理）

Producer 将消息发送到 Broker，Broker 负责将受到的消息存储到磁盘中，而 Consumer 负责从 Broker 订阅并消费消息。

Kafka 基本概念：

- **Producer**： 生产者，负责将消息发送到 Broker
- **Consumer**： 消费者，从 Broker 接收消息
- **Consumer Group**： 消费者组，由多个 Consumer 组成。消费者组内每个消费者负责消费不同分区的数据，**一个分区只能由一个组内消费者消费**，消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
- **Broker**： 可以看做一个独立的 **Kafka 服务节点或 Kafka 服务实例。**如果一台服务器上只部署了一个 Kafka 实例，那么我们也可以将 Broker 看做一台 Kafka 服务器。
- **Topic**： 一个逻辑上的概念，包含很多 Partition，**同一个 Topic 下的 Partiton 的消息内容是不相同的。**
- **Partition**： 为了实现扩展性，一个非常大的 topic **可以分布到多个 broker 上，一个 topic 可以分为多个 partition**，每个 partition 是一个有序的队列。
- **Replica**： 副本，**同一分区的不同副本保存的是相同的消息**，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，- kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，一个 leader 和若干个 follower。
- **Leader**： 每个分区的多个副本中的"主副本"，**生产者以及消费者只与 Leader 交互。**
- **Follower**： 每个分区的多个副本中的"从副本"，**负责实时从 Leader 中同步数据，保持和 Leader 数据的同步。**Leader 发生故障时，从 Follower 副本中重新选举新的 Leader 副本对外提供服务。

3. Kafka Replicas是怎么管理的?



- AR:分区中的**所有 Replica 统称为 AR**
- ISR:所有与 Leader 副本**保持一定程度同步**的Replica(包括 Leader 副本在内)组成 ISR
- OSR:与 Leader 副本**同步滞后过多的** Replica 组成了 OSR

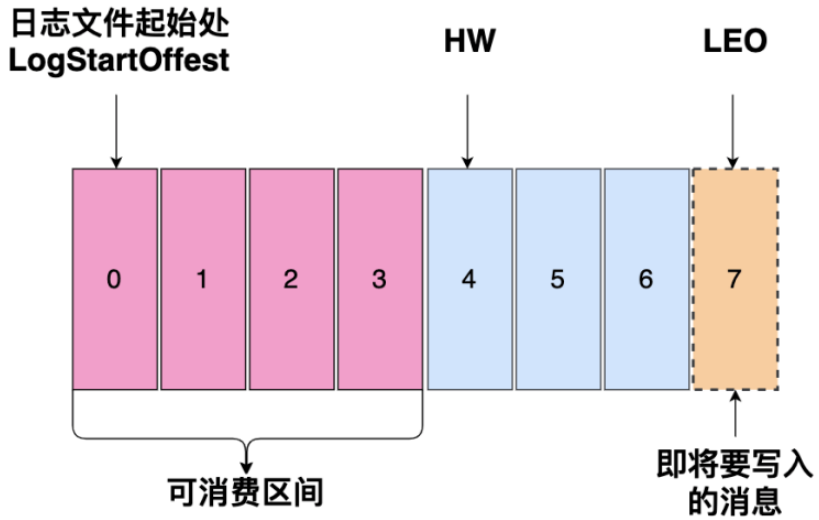
Leader 负责维护和跟踪 ISR 集合中所有 Follower 副本的滞后状态，当 Follower 副本落后过多时，就会将其放入 OSR 集合，当 Follower 副本追上了 Leader 的进度时，就会将其放入 ISR 集合。

默认情况下，只有 **ISR 中的副本才有资格晋升为 Leader。**

4. 如何确定当前能读到哪一条消息?

> 这个问题要先了解上一个问题的概念

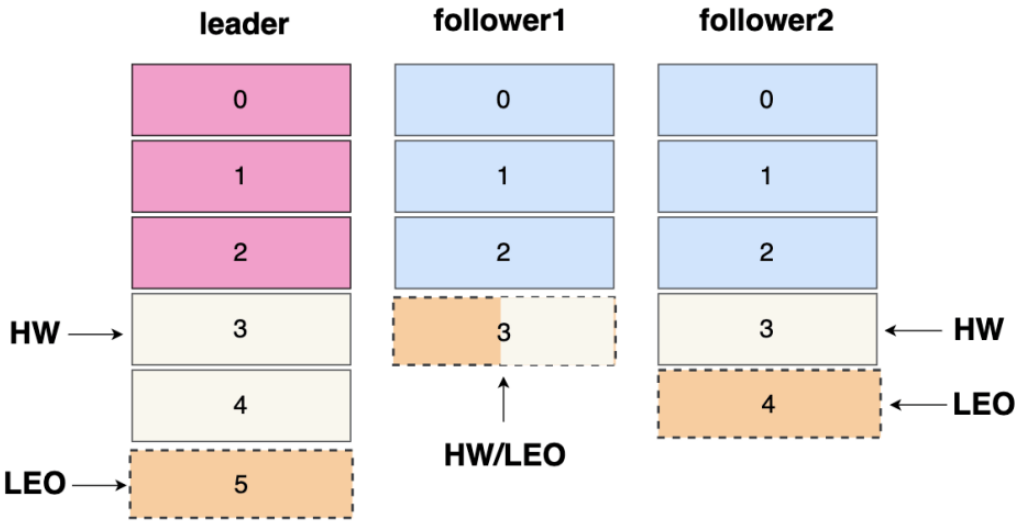
分区相当于一个日志文件，我们先简单介绍几个概念



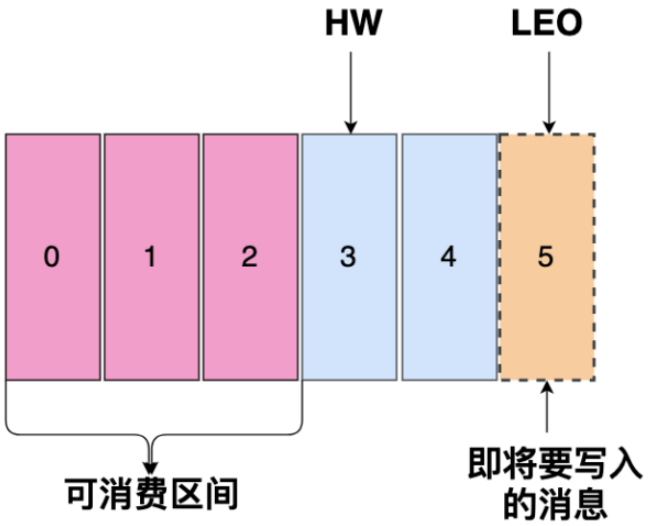
如上图是一个分区日志文件

- 标识**共有7条消息**，offset (消息偏移量)分别是0~6
- 0 代表这个日志文件的**开始**
- HW(High Watermark) 为4，0~3 代表这个日志文件**可以消费的区间**，消费者只能消费到这四条消息
- LEO 代表即将要写入消息的偏移量 offset

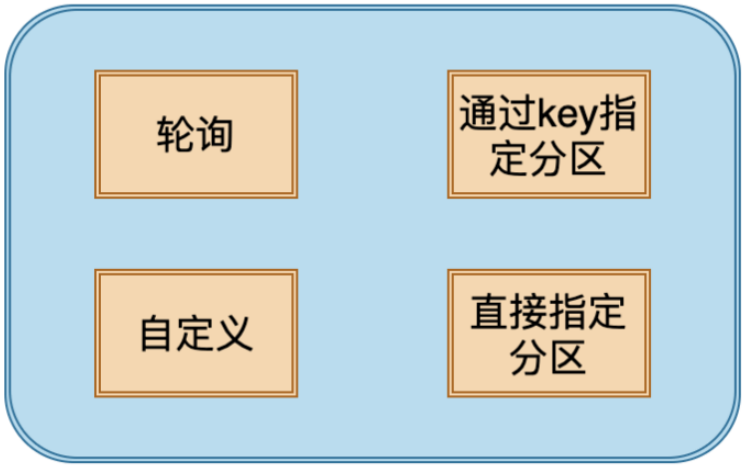
分区 ISR 集合中的每个副本都会维护自己的 LEO，而 ISR 集合中最小的LEO 即为分区的 HW



如上图: 三个分区副本都是 ISR集合当中的，最小的 LEO 为 3，就代表分区的 HW 为3，所以当前分区只能消费到 0~2 之间的三条数据，如下图



5. 发送消息的分区策略有哪些?



- 1.轮询：**依次**将消息发送该topic下的所有分区，如果在创建消息的时候 key 为 null，Kafka 默认采用这种策略。
- 2.key 指定分区：在创建消息是 key 不为空，并且使用默认分区器，Kafka 会将 key 进行 hash，然后**根据hash值映射到指定的分区上。**这样的好处是 key 相同的消息会在一个分区下，Kafka 并不能保证全局有序，但是在每个分区下的消息是有序的，按照顺序存储，按照顺序消费。在保证同一个 key 的消息是有序的，这样基本能满足消息的顺序性的需求。但是**如果 partation 数量发生变化，那就很难保证 key 与分区之间的映射关系了。**
- 3.自定义策略：实现 Partitioner 接口就能自定义分区策略。
- 4.指定 Partiton 发送

6. Kafka 的可靠性是怎么保证的?

1.acks

这个参数用来指定分区中有多少个副本收到这条消息，生产者才认为这条消息是写入成功的，这个参数有三个值：

- 1.acks = 1，默认为1。生产者发送消息，**只要 leader 副本成功写入消息，就代表成功。**这种方案的问题在于，当返回成功后，如果 leader 副本和 follower 副本**还没有来得及同步**，leader 就崩溃了，那么在选举后新的 leader 就没有这条**消息，也就丢失了。**

- 2.acks = 0。生产者发送消息后直接算写入成功，不需要等待响应。这个方案的问题很明显，**只要服务端写消息时出现任何问题，都会导致消息丢失。**
- 3.acks = -1 或 acks = all。生产者发送消息后，需要等待 ISR 中的所有副本都成功写入消息后才能收到服务端的响应。毫无疑问这种方案的**可靠性是最高的**，但是如果 ISR 中只有leader 副本，那么就和 acks = 1 毫无差别了。

2.消息发送的方式

第6问中我们提到了生产者发送消息有三种方式，发完即忘，同步和异步。我们可以通过同步或者异步获取响应结果，**失败做重试**来保证消息的可靠性。

3.手动提交位移

默认情况下，当消费者消费到消息后，就会自动提交位移。但是如果消费者消费出错，没有进入真正的业务处理，那么就可能会导致这条消息消费失败，从而丢失。我们可以开启手动提交位移，等待业务正常处理完成后，再提交offset。

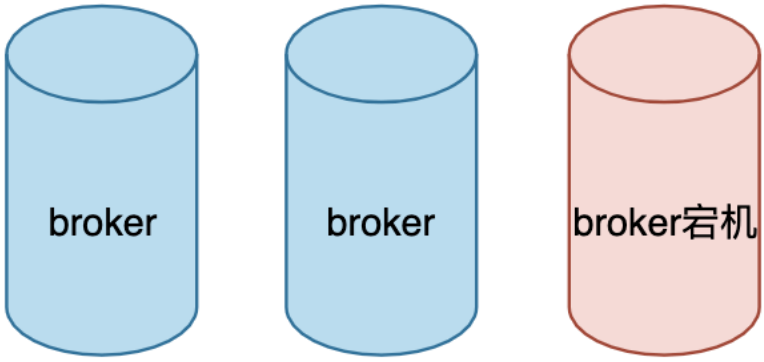
4.通过副本 LEO 来确定分区 HW

可参考第四问

7. 分区再分配是做什么的？解决了什么问题？

分区再分配主要是用来维护 kafka 集群的负载均衡

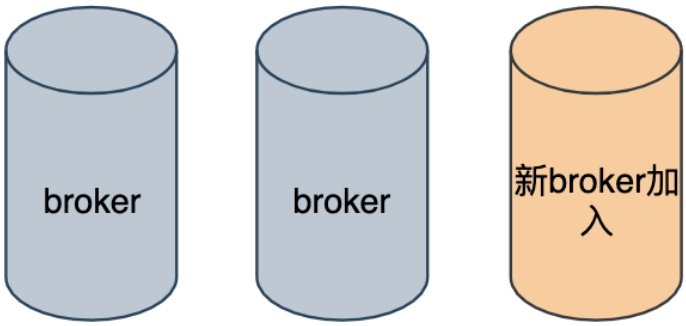
既然是分区再分配，那么 kafka 分区有什么问题呢？



问题1：当集群中的一个节点下线了

- 如果该节点的分区是单副本的,那么分区将会变得不可用
- 如果是多副本的，就会进行 leader 选举，在其他机器上选举出新的 leader

kafka 并不会将这些失效的分区迁移到其他可用的 broker 上，这样就会影响集群的负载均衡，甚至也会影响服务的可靠性和可用性



问题2：集群新增 broker 时，只有新的主题分区会分配在该 broker 上，而老的主题分区不会分配在该 broker 上，就造成了**老节点和新节点之间的负载不均衡。**

为了解决该问题就出现了分区再分配，它可以在集群扩容，broker 失效的场景下进行分区迁移。

分区再分配的原理就是通化控制器给分区新增新的副本，然后通过网络把旧的副本数据复制到新的副本上，在复制完成后，将旧副本清除。 当然，为了不影响集群正常的性能，在此复制期间还会有一系列保证性能的操作，比如**复制限流。**

8. Kafka Partition 副本 leader 是怎么选举的?

> 这个问题设计的点比较多，拓展的也更多一点，建议耐心阅读。

常用选主机制的缺点：

```
split-brain （脑裂）:
•这是由ZooKeeper的特性引起的，虽然ZooKeeper能保证所有Watch按顺序触发，但是网络延迟，并不能保证同一时刻所有Replica“看”到的状态是一样的，这就可能造成不同Replica的响应不一致，可能选出多个领导“大脑”，导致“脑裂”。

herd effect （羊群效应）:
•如果宕机的那个Broker上的Partition比较多， 会造成多个Watch被触发，造成集群内大量的调整，导致大量网络阻塞。

ZooKeeper负载过重:
•每个Replica都要为此在ZooKeeper上注册一个Watch，当集群规模增加到几千个Partition时ZooKeeper负载会过重。
```

优势：

Kafka的Leader Election方案解决了上述问题，它在所有broker中选出一个controller，所有Partition的Leader选举都由controller决定。 controller会将Leader的改变直接通过RPC的方式(比ZooKeeper Queue的方式更高效)通知需为此作为响应的Broker。

没有使用 zk，所以无 2.3 问题；也没有注册 watch无 2.2 问题 leader 失败了，就通过 controller 继续重新选举即可，所以克服所有问题。

Kafka partition leader的选举：

由 controller 执行：

- 从Zookeeper中读取当前分区的所有ISR(in-sync replicas)集合
- 调用配置的分区选择算法选择分区的leader

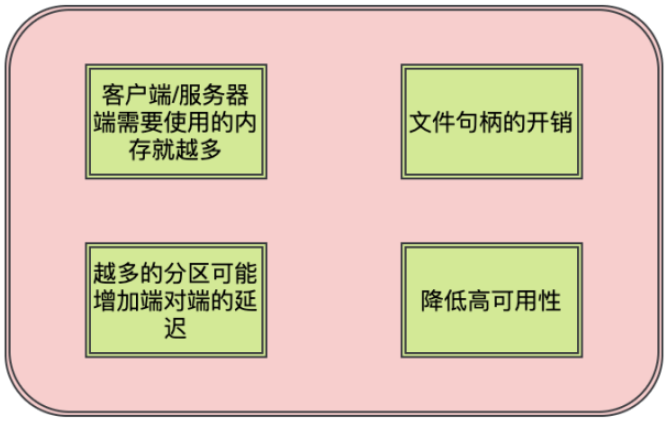
上面五种分区算法都是选择PreferredReplica(优先副本选举)作为当前Partition的leader。区别仅仅是选择leader之后的操作有所不同。

9. 分区数越多越好吗？吞吐量就会越高吗？

般类似于这种问题的答案，都是持否定态度的。

但是可以说，**在一定条件下，分区数的数量是和吞吐量成正比的，分区数和性能也是成正比的。**

那么为什么说超过了一定限度，就会对性能造成影响呢？原因如下:



1.客户端/服务器端需要使用的内存就越多

服务端在很多组件中都维护了分区级别的缓存，分区数越大，缓存成本也就越大。
消费端的消费线程数是和分区数挂钩的，分区数越大消费线程数也就越多，线程的开销成本也就越大
生产者发送消息有缓存的概念，会为每个分区缓存消息，当积累到一定程度或者时间时会将消息发送到分区，分区越多，这部分的缓存也就越大

2.文件句柄的开销

每个 partition 都会对应磁盘文件系统的一个目录。在 Kafka 的数据日志文件目录中，每个日志数据段都会分配两个文件，一个索引文件和一个数据文件。每个 broker 会为每个日志段文件打开一个 index 文件句柄和一个数据文件句柄。因此，随着 partition 的增多，所需要保持打开状态的文件句柄数也就越多，最终可能超过底层操作系统配置的文件句柄数量限制。

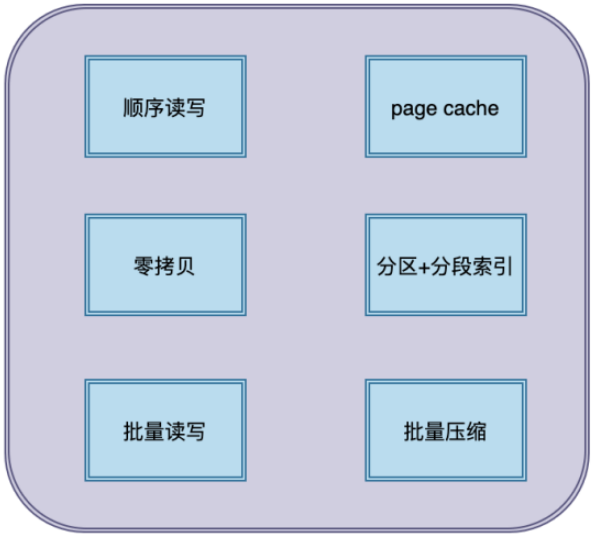
3.越多的分区可能增加端对端的延迟

Kafka 会将分区 HW 之前的消息暴露给消费者。分区越多则副本之间的同步数量就越多，在默认情况下，每个 broker 从其他 broker 节点进行数据副本复制时，该 broker 节点只会为此工作分配一个线程，该线程需要完成该 broker 所有 partition 数据的复制。

4.降低高可用性

在第 7 问我们提到了分区再分配，会将数据复制到另一份副本当中，**分区数量越多，那么恢复时间也就越长**，而如果发生宕机的 broker 恰好是 controller 节点时：在这种情况下，新 leader 节点的选举过程在 controller 节点恢复到新的 broker 之前不会启动。controller 节点的错误恢复将会自动地进行，但是新的 controller 节点需要从 zookeeper 中读取每一个 partition 的元数据信息用于初始化数据。例如，假设一个Kafka 集群存在 10000个partition，从 zookeeper 中恢复元数据时每个 partition 大约花费 2 ms，则 controller 的恢复将会增加约 20 秒的不可用时间窗口。

10. kafka 为什么这么快?



- 1.**顺序读写**磁盘分为顺序读写与随机读写，基于磁盘的随机读写确实很慢，但磁盘的顺序读写性能却很高，kafka 这里采用的就是顺序读写。
- 2.**Page Cache**为了优化读写性能，Kafka 利用了**操作系统本身的 Page Cache**，就是利用操作系统自身的内存而不是JVM空间内存。
- 3.**零拷贝**Kafka使用了零拷贝技术，也就是**直接将数据从内核空间的读缓冲区直接拷贝到内核空间的 socket 缓冲区**，然后再写入到 NIC 缓冲区，避免了在内核空间和用户空间之间穿梭。
- 4.**分区分段+索引**Kafka 的 message 是按 topic分 类存储的，topic 中的数据又是按照一个一个的 partition 即分区存储到不同 broker 节点。每个 partition 对应了操作系统上的一个文件夹，partition 实际上又是按照segment分段存储的。通过这种分区分段的设计，Kafka 的 message 消息实际上是分布式存储在一个一个小的 segment 中的，每次文件操作也是直接操作的 segment。为了进一步的查询优化，Kafka 又默认为分段后的数据文件建立了索引文件，就是文件系统上的.index文件。这种分区分段+索引的设计，不仅提升了数据读取的效率，同时也提高了数据操作的并行度。
- 5.**批量读写Kafka 数据读写也是批量的而不是单条的**,这样可以避免在网络上频繁传输单个消息带来的延迟和带宽开销。假设网络带宽为10MB/S，一次性传输10MB的消息比传输1KB的消息10000万次显然要快得多。
- 6.**批量压缩**Kafka 把所有的消息都变成一个**批量的文件**，并且进行合理的**批量压缩**，减少网络 IO 损耗，通过 mmap 提高 I/O 速度，写入数据的时候由于单个Partion是末尾添加所以速度最优；读取数据的时候配合 sendfile 进行直接读取。

低谷蓄力

MyBatis

1. 什么是MyBatis

> 这个问题主要是对比JDBC来看

1. MyBatis是一个ORM（对象关系映射）框架，它内部封装了JDBC,开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动，创建连接，创建statement等复杂的过程。开发人员不需要编写原生态sql，可以严格控制sql执行性能，灵活度高。
2. MyBatis可以使用xml或者注解来配置映射原生信息，将POJO映射成数据库中的记录，避免了几乎所有的JDBC代码和手动设置的参数以及获取结果集。

2. MyBatis的优点

1. 基于SQL语句编程，相对灵活（相对于hibernate），支持写动态sql语句并可重复使用。
2. 减少代码量，消除了冗余代码。（类似于JDBC的封装）
3. 与Spring完美集成。
4. 提供映射标签支持字段关系映射。

3. #{}和${}的区别是什么?

1. `#{}`预编译处理、是占位符，`${}`是字符串替换、是拼接符。
2. 使用`#{}`可以有效的防止sql注入，提高系统的安全性。

Mybatis在处理`#{}`的时候会将sql中的`#{}`替换成?号，调用PreparedStatement来赋值

```
/* SQL */
如：select * from user where name = #{userName}；设userName=javapub

看日志我们可以看到解析时将#{userName}替换成了 ？

select * from user where name = ?;

然后再把 javapub 放进去，外面加上单引号
```

Mybatis在处理`${}`的时候就是把`${}`替换成变量的值，调用Statement来赋值

```
/* SQL */
如：select * from user where name = #{userName}；设userName=javapub

看日志可以发现就是直接把值拼接上去了

select * from user where name = javapub;

这极有可能发生sql注入，下面举了一个简单的sql注入案例
```

4. 一个 Xml 映射文件，都会写一个 Dao 接口与之对应，这个 Dao 接口的工作原理是什么?

Dao 接口就是人们常说的 Mapper 接口，接口的全限名，就是映射文件中的 namespace 的值，接口的方法名就是映射文件中 **MappedStatement** 的 id 值，接口方法内的参数就是传递给 sql 的参数。

接口里的方法是**不能重载**的，因为是**全限名+方法名**的保存和寻找策略。

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

MappedStatement：MappedStatement维护了一条 <select|update|delete|insert>节点的封装,包括了传入参数映射配置、执行的SQL语句、结果映射配置等信息。

```
<select id="selectAuthorLinkedHashMap" resultType="java.util.LinkedHashMap">
        select id, username from author where id = #{value}
</select>
```

5. 如何获取自动生成的(主)键值?

用法:

在 `<insert />` 标签中添加 `useGeneratedKeys="true"` 等属性

```
<insert id="insert" useGeneratedKeys="true" keyProperty="id" keyColumn="id"
        parameterType="person" >
    INSERT INTO person(name, pswd)
    VALUE (#{name}, #{pswd})
</insert>
```

当 Mybatis 解析 xml节点时，读到 `insert` 有配置时，会判断是否 有配置 `useGeneratedKeys`，如果有则会使用 `Jdbc3KeyGenerator` 作为sql回显，否则会以 `NoKeyGenerator` 作为主键回显。

底层封装了JDBC获取自增主键，即当使用 prepareStatement 或者 Statement时候，可以通过 `getGeneratedKeys` 获取这条插入语句的自增而成的主键。例子

```
    Connection conn = DriverManager.getConnection(url, "root", "123456");
    String[] columnNames = {"id", "name"};
    PreparedStatement stmt = conn.prepareStatement(sql, columnNames);
    stmt.setString(1, "jack wang");
    stmt.executeUpdate();
    ResultSet rs = stmt.getGeneratedKeys();
    int id = 0;
    if (rs.next()) {
        id = rs.getInt(1);
        System.out.println("----------" + id);
    }
```

6. Mybatis 动态 sql 有什么用？有哪些动态 sql？执行原理?

Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。

Mybatis 提供了9种动态sql标签： **trim | where | set | foreach | if | choose | when | otherwise | bind。**

其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

是不是有点懵，继续阅读:

科普:

OGNL 是 Object-Graph Navigation Language 的缩写，对象图导航语言。例如 `#{}` 语法。

OGNL 作用是在对象和视图之间做数据的交互，可以存取对象的属性和调用对象的方法，通过表达式可以迭代出整个对象的结构图。

参考一个很形象的例子。

有一个学生对象 student，属性分别有 id = 10，name = '小明' 和 课程对象 course，其中 course 对象中属性有：分数 score = 88，排名 rank = 5。

对象关系图如下:

```
student

    id：10

    name：小明

    course：

        score：88

        rank：5
```

当上下文（环境）中的对象为 student 的时候，也就是在 Mybatis 中查询时传入的参数对象为 student 的时候：

通过 OGNL 表达式直接获取上下文中对象的属性值，比如：

#{id} —> 10，相对于当前上下文对象.getId()，即 student.getId() 。

#{name} —> 小明。

#{course.score} —> 88，相当于 student.getCourse().getScore()。

所以，通过 OGNL 表达式，可以迭代出整个对象的结构图。

发布 《最少必要面试题》

7. 什么是Mybatis的一级、二级缓存?

一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认一级缓存是开启的。

当Mybaits与Spring整合的时候，不带Spring事务的方法内，每次请求数据库，都会新建一个SqlSession，这时候是使用不到一级缓存的。除了事务问题，还有调用了Sqlsession的修改、添加、删除、commit()、close()等方法时，一级缓存也会被清空。

二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)。即使开启了二级缓存，不同的sqlsession之间的缓存数据也不是想互访就能互访的，必须等到sqlsession关闭了以后，才会把其一级缓存中的数据写入二级缓存。默认不打开二级缓存。

现在大多数应用都是支持分布式的，一般情况都是用中间件作为缓存层，比如redis。开启 MyBatis 的二级缓存也会多一步序列化和反序列化，影响服务性能。

8. MyBatis的工作原理

一图胜千文

1. 读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。
2. 加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在 MyBatis 配置文件 mybatis-config.xml 中加载。mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。
3. 构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。
4. 创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。

5. Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。
6. MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信息。
7. 输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过程。
8. 输出结果映射：输出结果类型可以是 Map、 List 等集合类型，也可以是基本数据类型和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

9. 什么是MyBatis的接口绑定？有哪些实现方式?

接口绑定，就是在 MyBatis 中任意定义接口，然后把接口里面的方法和SQL语句绑定，我们直接调用接口方法就可以，这样比起原来的SqlSession提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式：

- 通过**注解绑定**，就是在接口的方法上面加上 @Select、@Update 等注解，里面包含Sql语句来绑定；
- 通过**xml**里面写SQL来绑定， 在这种情况下，要指定xml映射文件里面的 namespace 必须为接口的全路径名。当Sql语句比较简单时候，用注解绑定， 当SQL语句比较复杂时候，用xml绑定，一般用xml绑定的比较多。

10. Mybatis的分页原理

Mybatis 使用 RowBounds 对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页，所以一般不会使用。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的原理就是使用 MyBatis 提供的插件接口，实现自定义插件，在插件的拦截方法内，拦截待执行的SQL，然后根据设置的 dialect（方言），和设置的分页参数，重写SQL，生成带有分页语句的 SQL，执行重写后的SQL，从而实现分页。

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

MySQL

推荐一篇非常不错的文章，阅读后更有利于了解MySQL【B树和B+树的区别】：
https://mp.weixin.qq.com/s/RWkc2lNarKnn8Dc0HrP58g

---

1. mysql有哪几种log

重做日志(redo log)、回滚日志(undo log)、二进制日志(binlog)、错误日志(errorlog)、慢查询日志(slow query log)、一般查询日志(general log)，中继日志(relay log)

错误日志：记录出错信息，也记录一些警告信息或者正确的信息。

查询日志：记录所有对数据库请求的信息，不论这些请求是否得到了正确的执行。

慢查询日志：设置一个阈值，将运行时间超过该值的所有SQL语句都记录到慢查询的日志文件中。

二进制日志：记录对数据库执行更改的所有操作。

中继日志：中继日志也是二进制日志，用来给slave 库恢复

事务日志：重做日志redo和回滚日志undo

---

2. MySQL的复制原理以及流程

1. 主：binlog线程——记录下所有改变了数据库数据的语句，放进master上的binlog中。
2. 从：io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进 自己的relay log 中。
3. 从：sql执行线程——执行relay log中的语句。

---

3. 事物的4种隔离级别

隔离强度逐渐增强，性能逐渐变差。

- 读未提交(RU) READ UNCOMMITTED
- 读已提交(RC) READ COMMITT
- 可重复读(RR) REPEATABLE READ
- 串行化 SERIALIZABLE

事务具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）四个特性，简称 ACID，缺一不可。

---

4. 相关概念

脏读

脏读指的是读到了其他事务未提交的数据，未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。读到了并一定最终存在的数据，这就是脏读。

可重复读

可重复读指的是在一个事务内，最开始读到的数据和事务结束前的任意时刻读到的同一批数据都是一致的。通常针对数据更新（UPDATE）操作。

不可重复读

对比可重复读，不可重复读指的是在同一事务内，不同的时刻读到的同一批数据可能是不一样的，可能会受到其他事务的影响，比如其他事务改了这批数据并提交了。通常针对数据更新（UPDATE）操作。

幻读

幻读是针对数据插入（INSERT）操作来说的。假设事务 A 对某些行的内容作了更改，但是还未提交，此时事务 B 插入了与事务 A 更改前的记录相同的记录行，并且在事务 A 提交之前先提交了，而这时，在事务 A 中查询，会发现好像刚刚的更改对于某些数据未起作用，但其实是事务 B 刚插入进来的，让用户感觉很魔幻，感觉出现了幻觉，这就叫幻读。

---

5. MySQL数据库几个基本的索引类型

普通索引、唯一索引、主键索引、全文索引

---

6. drop、delete与truncate的区

SQL中的drop、delete、truncate都表示删除，但是三者有一些差别

1、delete和truncate只删除表的数据不删除表的结构
2、速度,一般来说: drop> truncate >delete
3、delete语句是dml,这个操作会放到rollback segement中,事务提交之后才生效;
4、如果有相应的trigger,执行的时候将被触发. truncate,drop是ddl, 操作立即生效,原数据不放到rollback segment中,不能回滚. 操作不触发trigger.

---

7. 数据库的乐观锁和悲观锁是什么?

悲观锁的特点是先获取锁，再进行业务操作，即 “悲观” 的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常所说的 “一锁二查三更新” 即指的是使用悲观锁。

通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的 select … for update 操作来实现悲观锁。　当数据库执行 select for update 时会获取被 select 中的数据行的行锁，因此其他并发执行的 select for update 如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update 获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

mysql 还有个问题是 select… for update 语句执行中，如果数据表没有添加索引或主键，所有扫描过的行都会被锁上，这一点很容易造成问题。因此如果在 mysql 中用悲观锁务必要确定走了索引，而不是全表扫描。

乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号，或者时间戳。

乐观锁的两种实现方式:

1. 使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将 version 字段的值一同读出，数据每更新一次，对此 version 值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的 version 值进行比对，如果数据库表当前版本号与第一次取出来的 version 值相等，则予以更新，否则认为是过期数据。
2. 乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）,和上面的 version 类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则 OK，否则就是版本冲突。

---

8. SQL优化方式

1. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。
2. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如果索引是整形，那么可以在索引上设置默认值 0，确保表中列没有 null 值。
3. 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。
4. 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描。
5. in 和 not in 也要慎用，否则会导致全表扫描。
6. like ‘%abc%’ 也会导致全表扫描。
7. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。
8. 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。

9. 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
10. 很多时候用 exists 代替 in 是一个好的选择。

---

9. 从锁的类别上分MySQL都有哪些锁呢?

从锁的类别上来讲，有共享锁和排他锁。

共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。

排他锁: 又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

参考：

```
1. https://haicoder.net/note/mysql-interview/mysql-interview-optimistic-pessimism-lock.html
```

Redis

1. Redis是什么?

> 一般问这个问题你最少要答出以下几点

Redis 是一个**基于内存的 key-value** 存储系统，数据结构包括**字符串、list、set、zset（sorted set --有序集合）和hash**，bitmap，GeoHash(坐标)，HyperLogLog，Streams（5.x版本以后）

2. 你在哪些场景使用redis

> 你有实战经验，那就直接表演。如果没有，选几个下面的经典场景

1. 作为队列使用，（因为是基于内存、一般不会作为消费队列、作为循环队列必要适用）；
2. 模拟类似于token这种需要设置过期时间的场景，登录失效；
3. 分布式缓存，避免大量请求底层关系型数据库，大大降低数据库压力；
4. 分布式锁；
5. 基于 bitmap 实现布隆过滤器；
6. 排行榜-基于zset（有序集合数据类型）；
7. 计数器-对于浏览量、播放量等并发较高，使用 redis incr 实现计数器功能；
8. 分布式会话；
9. 消息系统；

3. 为什么Redis是单线程的?

这个问题给一个官方答案

因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。

4. Redis持久化有几种方式?

redis 提供了两种持久化的方式，分别是**快照方式（RDB Redis DataBase）和文件追加（AOF Append Only File）**。

显而易见，快照方式重启恢复快、但是数据更容易丢失，文件追加数据更完整、重启恢复慢。

混合持久化方式，Redis 4.0之后新增的方式，混合持久化是结合RDB和AOF的优点，在写入的时候先把当前的数据以RDB的形式写入到文件的开头，再将后续的操作以AOF的格式存入文件当中，这样既能保证重启时的速度，又能降低数据丢失的风险。

在恢复时，先恢复快照方式保存的文件，然后再恢复追加文件中的增量数据。

5. 什么是缓存穿透？怎么解决?

缓存穿透是指用户请求的数据在缓存中不存在即没有命中，同时在数据库中也不存在，导致用户每次请求该数据都要去数据库中查询一遍，然后返回空。

如果有恶意攻击者不断请求系统中不存在的数据，会导致短时间大量请求落在数据库上，造成数据库压力过大，甚至击垮数据库系统。

这就叫做缓存穿透。

怎么解决?

- 对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert之后清理缓存。
- 对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该Bitmap过滤。(也就是布隆过滤器的原理：大白话讲解布隆过滤器)

6. 什么是缓存雪崩?

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，请求直接落到数据库上，引起数据库压力过大甚至宕机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

怎么解决?

常用的解决方案有：

- 均匀过期
- 加互斥锁
- 缓存永不过期
- 双层缓存策略

均匀过期：设置不同的过期时间，让缓存失效的时间点尽量均匀。通常可以为有效期增加随机值或者统一规划有效期。

加互斥锁：跟缓存击穿解决思路一致，同一时间只让一个线程构建缓存，其他线程阻塞排队。

缓存永不过期：跟缓存击穿解决思路一致，缓存在物理上永远不过期，用一个异步的线程更新缓存。

双层缓存策略：使用主备两层缓存：

主缓存：有效期按照经验值设置，设置为主读取的缓存，主缓存失效后从数据库加载最新值。

备份缓存：有效期长，获取锁失败时读取的缓存，主缓存更新时需要同步更新备份缓存。

7. Redis使用上如何做内存优化?

1. **缩短键值的长度**

- 缩短值的长度才是关键，如果值是一个大的业务对象，可以将对象序列化成二进制数组；
- 首先应该在业务上进行精简，去掉不必要的属性，避免存储一些没用的数据；
- 其次是序列化的工具选择上，应该选择更高效的序列化工具来降低字节数组大小；
- 以JAVA为例，内置的序列化方式无论从速度还是压缩比都不尽如人意，这时可以选择更高效的序列化工具，如: protostuff，kryo等

2. **共享对象池**

对象共享池指Redis内部维护[0-9999]的整数对象池。创建大量的整数类型redisObject存在内存开销，每个redisObject内部结构至少占16字节，甚至超过了整数自身空间消耗。所以Redis内存维护一个[0-9999]的整数对象池，用于节约内存。 除了整数值对象，其他类型如list,hash,set,zset内部元素也可以使用整数对象池。因此开发中在满足需求的前提下，尽量使用整数对象以节省内存。

3. **字符串优化**

因为redis的惰性删除机制，字符串缩减后的空间不释放，作为预分配空间保留。尽量做新增不做更新。

4. **编码优化**

所谓编码就是具体使用哪种底层数据结构来实现。编码不同将直接影响数据的内存占用和读写效率。

这个需要掌握redis底层的数据结构。下图作为参考：

5. **控制key的数量**

8. 你们redis使用哪种部署方式?

redis部署分为单节点、主从部署（master-slave）、哨兵部署（Sentinel）、集群部署（cluster）。

单节点：也就是单机部署；

主从部署：分为一主一从或一主多从，主从之间同步分为全量或增量。量同步：master 节点通过 BGSAVE 生成对应的RDB文件，然后发送给slave节点，slave节点接收到写入命令后将master发送过来的文件加载并写入；增量同步：即在 master-slave 关系建立开始，master每执行一次数据变更的命令就会同步至slave节点。一般会将写请求转发到master，读请求转发到slave。提高了redis的性能。

哨兵部署：分别有哨兵集群与Redis的主从集群，哨兵作为操作系统中的一个监控进程，对应监控每一个Redis实例，如果master服务异常（ping pong其中节点没有回复且超过了一定时间），就会多个哨兵之间进行确认，如果超过一半确认服务异常，则对master服务进行下线处理，并且选举出当前一个slave节点来转换成master节点；如果slave节点服务异常，也是经过多个哨兵确认后，进行下线处理。提高了redis集群高可用的特性，及横向扩展能力的增强。

集群部署：属于**“去中心化”**的一种方式，多个 master 节点保存整个集群中的全部数据，而数据根据 key 进行 crc-16 校验算法进行散列，将 key 散列成对应 16383 个 slot，而 Redis cluster 集群中每个 master 节点负责不同的slot范围。每个 master 节点下还可以配置多个 slave 节点，同时也可以在集群中再使用 sentinel 哨兵提升整个集群的高可用性。

9. redis实现分布式锁要注意什么?

1. 加锁过程要保证原子性;
2. 保证谁加的锁只能被谁解锁，即Redis加锁的value，解锁时需要传入相同的value才能成功，保证value唯一性;
3. 设置锁超时时间，防止加锁方异常无法释放锁时其他客户端无法获取锁，同时，超时时间要大于业务处理时间;

使用Redis命令 `SET lock_key unique_value NX EX seconds` 进行加锁，单命令操作，Redis是串行执行命令，所以能保证只有一个能加锁成功。

Spring

1. 什么是 Spring 框架？Spring 框架有哪些主要模块?

Spring是针对bean的生命周期进行管理的轻量级容器，一个控制反转和面向切面的容器框架

Spring有七大功能模块:

1、Core

Core模块是Spring的核心类库，Core实现了IOC功能。

2、AOP

Apring AOP模块是Spring的AOP库，提供了AOP（拦截器）机制，并提供常见的拦截器，供用户自定义和配置。

3、orm

提供对常用ORM框架的管理和支持，hibernate、mybatis等。

4、Dao

Spring提供对JDBC的支持，对JDBC进行封装。

5、Web

对Struts2的支持。

6、Context

Context模块提供框架式的Bean的访问方式，其它程序可以通过Context访问Spring的Bean资源，相当于资源注入。

7、MVC

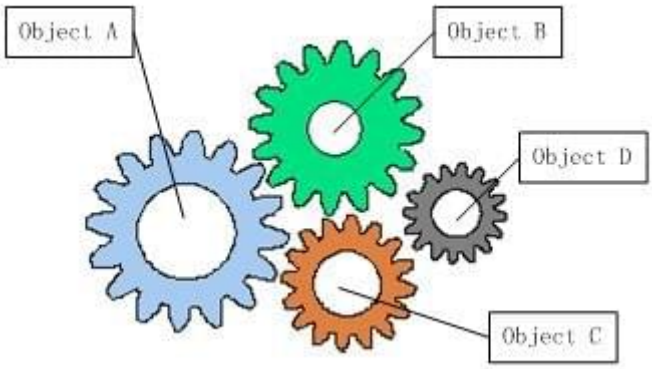
MVC模块为spring提供了一套轻量级的MVC实现，即Spring MVC。

2. Spring IOC、AOP举例说明

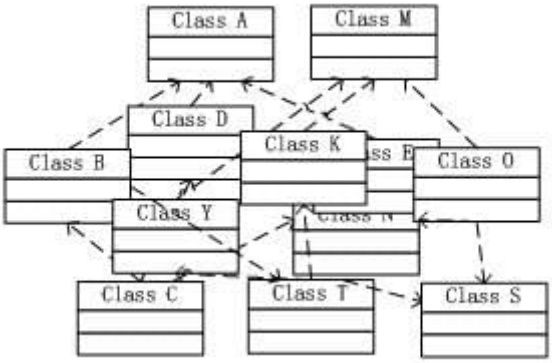
> 这是一个基础问题，如果理解有难度、建议先读五遍。

1、IOC理论的背景

我们都知道，在采用面向对象方法设计的软件系统中，它的底层实现都是由N个对象组成的，所有的对象通过彼此的合作，最终实现系统的业务逻辑。



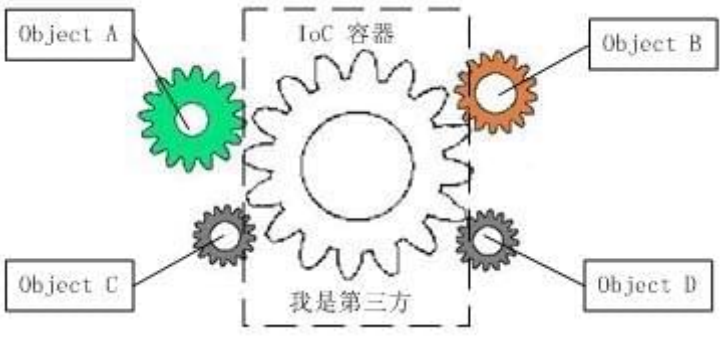
如果我们打开机械式手表的后盖，就会看到与上面类似的情形，各个齿轮分别带动时针、分针和秒 针顺时针旋转，从而在表盘上产生正确的时间。图1中描述的就是这样的一个齿轮组，它拥有多个独立的齿轮，这些齿轮相互啮合在一起，协同工作，共同完成某项 任务。我们可以看到，在这样的齿轮组中，如果有一个齿轮出了问题，就可能会影响到整个齿轮组的正常运转。
齿轮组中齿轮之间的啮合关系,与软件系统中对象之间的耦合关系非常相似。对象之间的耦合关系是无法避免的，也是必要的，这是协同工作的基础。现在，伴随着 工业级应用的规模越来越庞大，对象之间的依赖关系也越来越复杂，经常会出现对象之间的多重依赖性关系，因此，架构师和设计师对于系统的分析和设计，将面临 更大的挑战。对象之间耦合度过高的系统，必然会出现牵一发而动全身的情形。



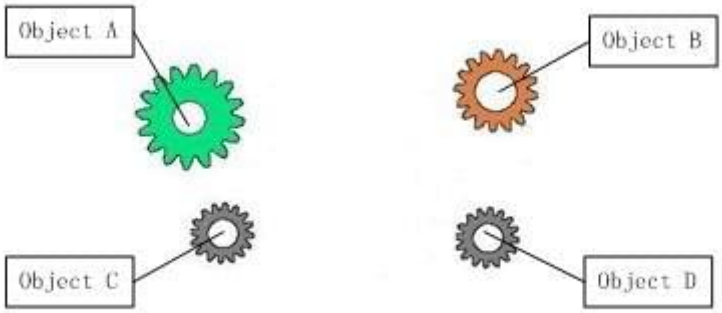
耦合关系不仅会出现在对象与对象之间，也会出现在软件系统的各模块之间，以及软件系统和硬件系统之间。如何降低系统之间、模块之间和对象之间的耦合度，是软件工程永远追求的目标之一。**为了解决对象之间的耦合度过高的问题，软件专家Michael Mattson提出了IOC理论，用来实现对象之间的“解耦”，目前这个理论已经被成功地应用到实践当中，很多的J2EE项目均采用了IOC框架产品Spring。**

2、什么是控制反转(IoC)

IOC是Inversion of Control的缩写，多数书籍翻译成“控制反转”，还有些书籍翻译成为“控制反向”或者“控制倒置”。
1996年，Michael Mattson在一篇有关探讨面向对象框架的文章中，首先提出了IOC 这个概念。对于面向对象设计及编程的基本思想，前面我们已经讲了很多了，不再赘述，简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以 后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC理论提出的观点大体是这样的：借助于“第三方”实现具有依 赖关系的对象之间的解耦，如下图：



大家看到了吧，由于引进了中间位置的“第三方”，也就是IOC容器，使得A、B、C、D这4个对象没有了耦合关系，齿轮之间的传动全部依靠“第三 方”了，全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有 对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把IOC容器比喻成“粘合剂”的由来。
我们再来做个试验：把上图中间的IOC容器拿掉，然后再来看看这套系统：



我们现在看到的画面，就是我们要实现整个系统所需要完成的全部内容。这时候，A、B、C、D这4个对象之间已经没有了耦合关系，彼此毫无联系，这样 的话，当你在实现A的时候，根本无须再去考虑B、C和D了，对象之间的依赖关系已经降低到了最低程度。所以，如果真能实现IOC容器，对于系统开发而言， 这将是一件多么美好的事情，参与开发的每一成员只要实现自己的类就可以了，跟别人没有任何关系！
我们再来看看，控制反转(IOC)到底为什么要起这么个名字？我们来对比一下：
软件系统在没有引入IOC容器之前，如图1所示，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。
软件系统在引入IOC容器之后，这种情形就完全改变了，如图3所示，由于IOC容器的加入，对象A与对象B之间失去了直接联系，所以，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方。
通过前后的对比，我们不难看出来：对象A获得依赖对象B的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

参考资料：https://www.cnblogs.com/jianmang/articles/4947615.html

3. 什么是控制反转(IOC)？什么是依赖注入（DI）？

IoC(Inversion of Control) – 控制反转。它不是一种技术，而是一种思想。

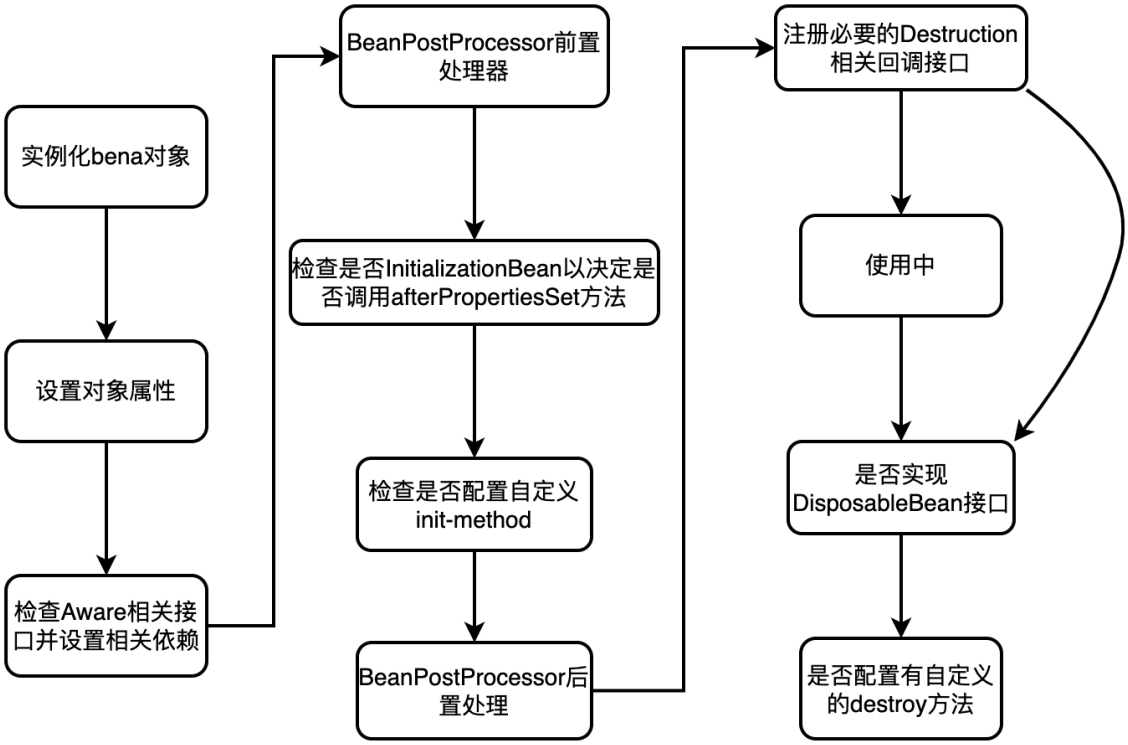
IOC：就是对象之间的依赖关系由容器来创建，对象之间的关系本来是由我们开发者自己创建和维护的，在我们使用Spring框架后，对象之间的关系由容器来创建和维护，将开发者做的事让容器做，这就是控制反转。BeanFactory接口是Spring Ioc容器的核心接口。

DI：我们在使用Spring容器的时候，容器通过调用set方法或者是构造器来建立对象之间的依赖关系。控制反转是目标，依赖注入是我们实现控制反转的一种手段。

4. 描述一下 Spring Bean 的生命周期?

> 这道题是spring一道标准题目

按照阶段理解Spring中的bean的生命周期主要包含四个阶段：`实例化Bean --> Bean属性填充 --> 初始化Bean -->销毁Bean`



1. Spring中的bean的生命周期主要包含四个阶段：实例化Bean -- > Bean属性填充 -- > 初始化Bean -- > 销毁Bean
2. 首先是实例化Bean，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用doCreateBean()方法进行实例化，实际上就是通过反射的方式创建出一个bean对象
3. Bean实例创建出来后，接着就是给这个Bean对象进行属性填充，也就是注入这个Bean依赖的其它bean对象
4. 属性填充完成后，进行初始化Bean操作，初始化阶段又可以分为几个步骤：

a. 执行Aware接口的方法

> Spring会检测该对象是否实现了xxxAware接口，通过Aware类型的接口，可以让我们拿到Spring容器的些资源。如实现BeanNameAware接口可以获取到BeanName，实现BeanFactoryAware接口可以获取到工厂对象BeanFactory等

b. 执行BeanPostProcessor的前置处理方法postProcessBeforeInitialization()，对Bean进行一些自定义的前置处理

c. 判断Bean是否实现了InitializingBean接口，如果实现了，将会执行InitializingBean的afeterPropertiesSet()初始化方法；

d. 执行用户自定义的初始化方法，如init-method等；

e. 执行BeanPostProcessor的后置处理方法postProcessAfterinitialization()

5. 初始化完成后，Bean就成功创建了，之后就可以使用这个Bean， 当Bean不再需要时，会进行销毁操作，

a. 首先判断Bean是否实现了DestructionAwareBeanPostProcessor接口，如果实现了，则会执行DestructionAwareBeanPostProcessor后置处理器的销毁回调方法
b. 其次会判断Bean是否实现了DisposableBean接口，如果实现了将会调用其实现的destroy()方法
c. 最后判断这个Bean是否配置了dlestroy-method等自定义的销毁方法，如果有的话，则会自动调用其配置的销毁方法；

5. Spring Bean 的作用域之间有什么区别?

Spring器中的bean可以分为5个范围:

1. singleton：这种bean范围是默认的，这种范围确保不管接受多少请求，每个容器中只有一个bean的实例，单例模式；
2. prototype：为每一个bean提供一个实例；
3. request：在请求bean范围内为每一个来自客户端的网络请求创建一个实例，在请求完毕后，bean会失效并被垃圾回收器回收；
4. session：为每个session创建一个实例，session过期后，bean会随之消失；
5. global-session：global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet公用全局的存储变量的话，那么全局变量需要存储在global-session中。

6. Spring中都应用了哪些设计模式

1、简单工厂模式

简单工厂模式的本质就是一个工厂类根据传入的参数，动态的决定实例化哪个类。

Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象。

2、工厂方法模式

应用程序将对象的创建及初始化职责交给工厂对象，工厂Bean。

定义工厂方法，然后通过config.xml配置文件，将其纳入Spring容器来管理，需要通过factory-method指定静态方法名称。

3、单例模式

Spring用的是双重判断加锁的单例模式，通过getSingleton方法从singletonObjects中获取bean。

```
     /**
     * Return the (raw) singleton object registered under the given name.
     * <p>Checks already instantiated singletons and also allows for an early
     * reference to a currently created singleton (resolving a circular
reference).
     * @param beanName the name of the bean to look for
     * @param allowEarlyReference whether early references should be created or
not
     * @return the registered singleton object, or {@code null} if none found
     */
    protected Object getSingleton(String beanName, boolean allowEarlyReference)
{
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null && isSingletonCurrentlyInCreation(beanName))
{
            synchronized (this.singletonObjects) {
                singletonObject = this.earlySingletonObjects.get(beanName);
                if (singletonObject == null && allowEarlyReference) {
                    ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
                    if (singletonFactory != null) {
                        singletonObject = singletonFactory.getObject();
                        this.earlySingletonObjects.put(beanName,
singletonObject);
                        this.singletonFactories.remove(beanName);
```

```
                    }
                }
            }
        }
        return (singletonObject != NULL_OBJECT ? singletonObject : null);
    }
```

4、代理模式

Spring的AOP中，使用的Advice（通知）来增强被代理类的功能。Spring实现AOP功能的原理就是代理模式（① JDK动态代理，② CGLIB字节码生成技术代理。）对类进行方法级别的切面增强。

5、装饰器模式

装饰器模式：动态的给一个对象添加一些额外的功能。

Spring的ApplicationContext中配置所有的DataSource。这些DataSource可能是不同的数据库，然后SessionFactory根据用户的每次请求，将DataSource设置成不同的数据源，以达到切换数据源的目的。

在Spring中有两种表现：

一种是类名中含有Wrapper，另一种是类名中含有Decorator。

6、观察者模式

定义对象间的一对多的关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

Spring中观察者模式一般用在listener的实现。

7、策略模式

策略模式是行为性模式，调用不同的方法，适应行为的变化 ，强调父类的调用子类的特性 。

getHandler是HandlerMapping接口中的唯一方法，用于根据请求找到匹配的处理器。

8、模板方法模式

Spring JdbcTemplate的query方法总体结构是一个模板方法+回调函数，query方法中调用的execute()是一个模板方法，而预期的回调doInStatement(Statement state)方法也是一个模板方法。

7. Spring AOP里面的几个名词的概念

（1）连接点（Join point）：指程序运行过程中所执行的方法。在Spring AOP中，一个连接点总代表一个方法的执行。

（2）切面（Aspect）：被抽取出来的公共模块，可以用来会横切多个对象。Aspect切面可以看成Pointcut切点 和 Advice通知 的结合，一个切面可以由多个切点和通知组成。

> 在Spring AOP中，切面可以在类上使用 @AspectJ 注解来实现。

（3）切点（Pointcut）：切点用于定义 要对哪些Join point进行拦截。

> 切点分为execution方式和annotation方式。execution方式可以用路径表达式指定对哪些方法拦截，比如指定拦截add、*search*。annotation方式可以指定被哪些注解修饰的代码进行拦截。

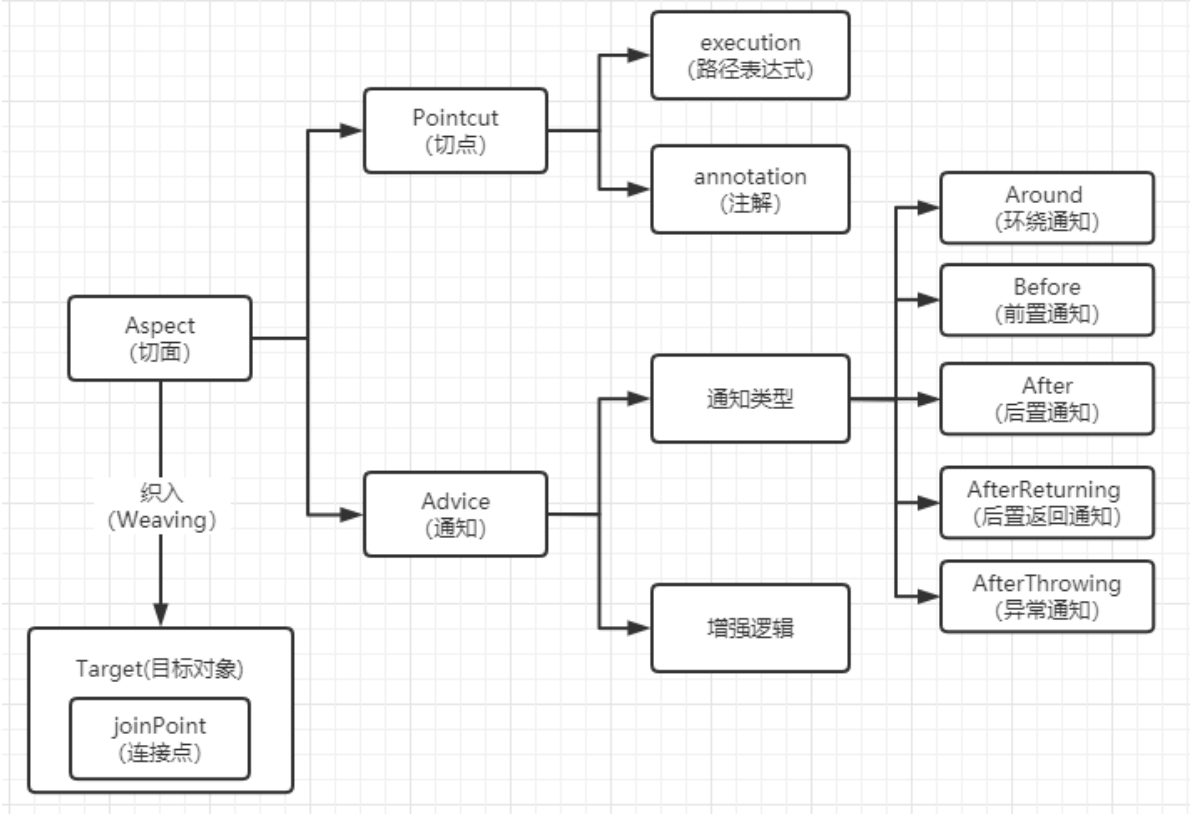
（4）通知（Advice）：指要在连接点（Join Point）上执行的动作，即增强的逻辑，比如权限校验和、日志记录等。通知有各种类型，包括Around、Before、After、After returning、After throwing。

（5）目标对象（Target）：包含连接点的对象，也称作被通知（Advice）的对象。 由于Spring AOP是通过动态代理实现的，所以这个对象永远是一个代理对象。

（6）织入（Weaving）：通过动态代理，在目标对象（Target）的方法（即连接点Join point）中执行增强逻辑（Advice）的过程。

（7）引入（Introduction）：添加额外的方法或者字段到被通知的类。Spring允许引入新的接口（以及对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

几个概念的关系图可以参考下图：



网上有张非常形象的图，描述了各个概念所处的场景和作用，贴在这里供大家理解：

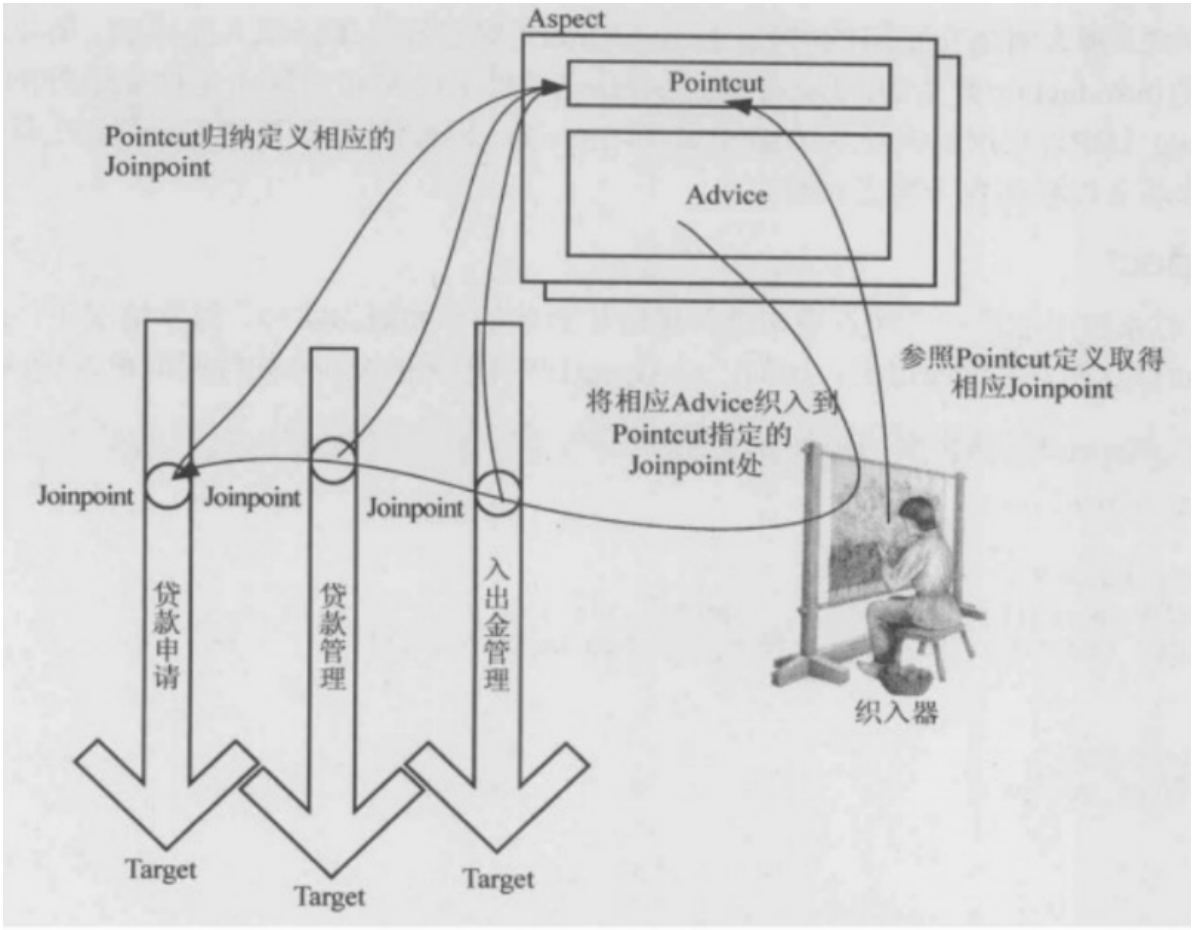


图7-9　AOP各个概念所处的场景

8. BeanFactory和ApplicationContext有什么区别?

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

1. BeanFactory是Spring里面最底层的接口，是IoC的核心，定义了IoC的基本功能，包含了各种Bean的定义、加载、实例化，依赖注入和生命周期管理。ApplicationContext接口作为BeanFactory的子类，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。
- 资源文件访问，如URL和文件（ResourceLoader）。
- 载入多个（有继承关系）上下文（即同时加载多个配置文件），使得每一个上下文都专注于一个特定的层次，比如应用的web层。
- 提供在监听器中注册bean的事件。

2. a. BeanFactroy采用的是延迟加载形式来注入Bean的，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，我们就不能提前发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

b. ApplicationContext，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，这样有利于检查所依赖属性是否注入。

c. ApplicationContext启动后预载入所有的单实例Bean，所以在运行的时候速度比较快，因为它们已经创建好了。相对于BeanFactory，ApplicationContext 唯一的不足是占用内存空间，当应用程序配置Bean较多时，程序启动较慢。

3. 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。
4. BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

9. Spring如何解决循环依赖问题：

见：https://javapub.blog.csdn.net/

循环依赖问题在Spring中主要有三种情况：

- （1）通过构造方法进行依赖注入时产生的循环依赖问题。
- （2）通过setter方法进行依赖注入且是在多例（原型）模式下产生的循环依赖问题。
- （3）通过setter方法进行依赖注入且是在单例模式下产生的循环依赖问题。

在Spring中，只有第（3）种方式的循环依赖问题被解决了，其他两种方式在遇到循环依赖问题时都会产生异常。这是因为：

- 第一种构造方法注入的情况下，在new对象的时候就会堵塞住了，其实也就是"先有鸡还是先有蛋"的历史难题。
- 第二种setter方法（多例）的情况下，每一次getBean()时，都会产生一个新的Bean，如此反复下去就会有无穷无尽的Bean产生了，最终就会导致OOM问题的出现。

Spring在单例模式下的setter方法依赖注入引起的循环依赖问题，主要是通过二级缓存和三级缓存来解决的，其中三级缓存是主要功臣。解决的核心原理就是：在对象实例化之后，依赖注入之前，Spring提前暴露的Bean实例的引用在第三级缓存中进行存储。

第一种构造方法注入的情况：

> 例如：类A通过构造函数注入需要类B的实例，而类B通过构造函数注入需要类A的实例。如果将A类和B类的bean配置为相互注入，则Spring IoC容器会在运行时检测此循环引用，并抛出a `BeanCurrentlyInCreationException`。
> 一种可能的解决方案是编辑由setter而不是构造函数配置的某些类的源代码。或者，避免构造函数注入并仅使用setter注入。换句话说，尽管不推荐使用，但您可以使用setter注入配置循环依赖关系。
> 与典型情况（没有循环依赖）不同，bean A和bean B之间的循环依赖强制其中一个bean在完全初始化之前被注入另一个bean（经典的鸡与鸡蛋场景）。

```
    <bean id="person" class="pojo.Person">
        <constructor-arg index="0" value="小明"/>
        <constructor-arg index="1" value="12"/>
        <constructor-arg index="2" value="student"/>
    </bean>

    <bean id="student" class="pojo.Student">
        <constructor-arg index="0" value="小王"/>
        <constructor-arg index="1" value="13"/>
        <constructor-arg index="2" value="person"/>
    </bean>
```

10. Spring事务的实现方式和实现原理：

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。Spring只提供统一事务管理接口，具体实现都是由各数据库自己实现，数据库事务的提交和回滚是通过 redo log 和 undo log实现的。Spring会在事务开始时，根据当前环境中设置的隔离级别，调整数据库隔离级别，由此保持一致。

1. Spring事务的种类：

spring支持编程式事务管理和声明式事务管理两种方式：

a. 编程式事务管理使用 TransactionTemplate。

b. 声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前启动一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

> 声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中，减少业务代码的污染。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

2. spring的事务传播机制：

spring事务的传播机制说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。事务传播机制实际上是使用简单的ThreadLocal实现的，所以，如果调用的方法是在新线程调用的，事务传播实际上是会失效的。

> ① PROPAGATION_REQUIRED：（默认传播行为）如果当前没有事务，就创建一个新事务；如果当前存在事务，就加入该事务。
>
> ② PROPAGATION_REQUIRES_NEW：无论当前存不存在事务，都创建新事务进行执行。
>
> ③ PROPAGATION_SUPPORTS：如果当前存在事务，就加入该事务；如果当前不存在事务，就以非事务执行。‘
>
> ④ 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。
>
> ⑤ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则按REQUIRED属性执行。
>
> ⑥ PROPAGATION_MANDATORY：如果当前存在事务，就加入该事务；如果当前不存在事务，就抛出异常。
>
> ⑦ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

事务不只限于脏读、幻读等名词。

3. Spring中的隔离级别：

> ① ISOLATION_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。
>
> ② ISOLATION_READ_UNCOMMITTED：读未提交，允许事务在执行过程中，读取其他事务未提交的数据。
>
> ③ ISOLATION_READ_COMMITTED：读已提交，允许事务在执行过程中，读取其他事务已经提交的数据。
>
> ④ ISOLATION_REPEATABLE_READ：可重复读，在同一个事务内，任意时刻的查询结果都是一致的。
>
> ⑤ ISOLATION_SERIALIZABLE：所有事务逐个依次执行。

低谷蓄力

SpringBoot

1. 为什么要用 spring boot?

通过自动配置方式简化 Spring 应用的开发，弱化配置，遵循 **约定大于配置的原则** ，使开发者专注于业务开发而无需过多考虑配置相关操作，通过启动类的 main 方法一键启动应用。

2. spring boot 有哪些优点?

1. 独立运行。
   内嵌了 servlet，tomat 等，不需要打成 war 包部署到容器中，只需要将 SpringBoot 项目打成jar包就能独立运行。
2. 简化配置。
   启动器自动依赖其他组件，简少了 maven 的配置。各种常用组件及配置已经默认配置完成，无需过多干预。
3. 避免大量的 Maven 导入和各种版本冲突。
4. 应用监控。
   Spring Boot 提供一系列端点可以监控服务及应用。

3. spring boot 核心配置文件是什么?

springboot 核心的两个配置文件：

- bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，
  比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效，且 bootstrap 里面的属性不能被覆盖；一般来说我们在 SpringCloud Config 或者 Nacos 中会用到它。
- application (. yml 或者 . properties)：用于 springboot 项目的自动化配置

4. spring boot的核心注解是什么？由那些注解组成?

核心注解为：**@SpringBootApplication**

该注解主要由三个注解组成：

`@SpringBootConfiguration()`:代表当前是一个配置类
`@EnableAutoConfiguration()`: 启动自动配置
`@ComponentScan()`： 指定扫描哪些 Spring 注解

5. 说一下springboot的自动装配原理

1. SpringBoot启动的时候加载主配置类，开启了自动配置功能@EnableAutoConfiguration。
2. 查看@EnableAutoConfiguration，其作用是利用AutoConfigurationImportSelector给容器中导入一些组件。
3. 查看AutoConfigurationImportSelector，其中public String[] selectImports(AnnotationMetadata annotationMetadata)方法内 最终调用getCandidateConfigurations()方法
4. 查看 getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes)，获取候选的配置，这个是扫描所有jar包类路径下"META-INF/spring.factories"
5. 然后把扫描到的这些文件包装成Properties对象。
6. 从properties中获取到EnableAutoConfiguration.class类名对应的值，然后把他们添加在容器中。

简而言之，整个过程就是将类路径下 "META-INF/spring.factories" 里面配置的所有 EnableAutoConfiguration 的值加入到容器中。

6. SpringBoot、Spring MVC和Spring有什么区别?

Spring：主要用来创建IOC容器，依赖注入，实现程序间的松耦合

SpringMVC: 主要是用来做WEB开发，通过各种组件的协调配合，简化Web应用的开发

SpringBoot: SpringBoot更像是一个管家，当使用到对应功能时，只需要导入指定应用启动器，SpringBoot就能够在底层默认其配置，大大简化了开发所需的繁杂配置

7. SpringBoot启动时都做了什么?

Springboot 的启动，主要创建了配置环境 (environment)、事件监听 (listeners)、应用上下文 (applicationContext)，并基于以上条件，在容器中开始实例化我们需要的 Bean，至此，通过 SpringBoot 启动的程序已经构造完成。

8. SpringBoot 中的监视器是什么?

SpringBoot Actuator 是 SpringBoot 一项重要功能，其可以帮助我们查看应用的运行状态，对运行时指标进行检查和监控，监视器提供了一组可以直接作为 httpurl 访问的 rest 端点来访问查看指定功能状态。

9. SpringBoot 中的starter到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。

正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

拓展：如何自定义starter?

1. 创建项目，创建两个模块分别为 `spring-boot-starter-*`，`spring-boot-starter-*-autoconfiguration`
2. `spring-boot-starter-*` pom 引入 `spring-boot-starter-*-autoconfiguration`
3. `spring-boot-starter-*-autoconfiguration` 创建功能方法，创建 *properties类，创建一个配置类将功能方法类添加到 spring 容器，在 resouces 下创建 `META-INF/spring.factories` 配置

   `org.springframework.boot.autoconfigure.EnableAutoConfiguration=\`

   `org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration`
4. 打包安装这两个模块到本地 maven 仓库，即可在其他项目引入 `spring-boot-starter-*` 注入功能类进行方法调用

10. 微服务中如何实现 session 共享?

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。
这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

Zookeeper

1. 什么是 Zookeeper

ZooKeeper 是一个开源的分布式协调服务。它是一个为分布式应用提供一致性服务的软件，分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 从设计模式角度来理解， 是一个基于**观察者模式**设计的分布式服务管理框架，它**负责存储和管理大家都关心的数据**，然后**接受观察者的注册**，一旦这些数据的状态发生变化，Zookeeper就将**负责通知已经在Zookeeper上注册的那些观察者**做出反应。

观察者模式是什么：设计模式

可以这样理解：

`ZooKeeper=文件系统+通知机制`

2. ZK 的节点类型

> 这道题相信大家都有所了解，zookeeper v3.6.2 版本后，支持7种节点类型。持久；持久顺序；临时；临时顺序；容器；持久 TTL；持久顺序 TTL。

说出这几种类型当然已经回答了问题，但是细节的描述更能体现你的知识底蕴。

持久 TTL、持久顺序 TTL

关于持久和顺序这两个关键字，不用我再解释了，这两种类型的节点重点是后面的 TTL，TTL 是 `time to live` 的缩写，指带有存活时间，简单来说就是当该节点下面没有子节点的话，超过了 TTL 指定时间后就会被自动删除，但是 TTL 启用是需要额外的配置(这个之前也有提过)配置是 zookeeper.extendedTypesEnabled 需要配置成 true，否则的话创建 TTL 时会收到 Unimplemented 的报错。

3. Zookeeper 下 Server 工作状态有哪些?

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

- **LOOKING**：寻 找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入 Leader 选举状态。
- **FOLLOWING**：跟随者状态。表明当前服务器角色是 Follower。
- **LEADING**：领导者状态。表明当前服务器角色是 Leader。
- **OBSERVING**：观察者状态。表明当前服务器角色是 Observer。（Observer角色除了不能投票(以及和投票相关的能力)和过半写成功策略外，其它和follower功能一样。observer角色减轻了投票的压力，在以前通过增、减follower的数量提高伸缩性。投票来说，follower是有状态的，都直接影响投票结果，特别是follower的数量越多，投票过程的性能就越差。）

4. zookeeper是cp还是ap?

zk遵循的是CP原则，即保证一致性和网络分区容错性，但不保证可用性。

什么是cap?

Consistency（一致性）：分布式系统中多个主机之间是否能够保持数据一致性的特性。即当系统数据发生更新操作之后，各个主机中的数据是否仍然处于一致的状态。

Availability（可用性）：系统提供的服务必须一直处于可用的状态，即对于的每一个请求，系统总是可以在**有限的时间**内对用户做出响应。

Partition tolerance（分区容错性）：分布式系统在遇到任何网络分区故障时候，仍然保证对外提供满足一致性和可用性的服务。

5. 说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

6. 介绍一下ZAB协议?

ZAB协议是为分布式协调服务Zookeeper专门设计的一种支持崩溃恢复的原子广播协议。

ZAB协议包括两种基本的模式：

1. 崩溃恢复
2. 消息广播

当整个 zookeeper 集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

7. ZAB 和 Paxos 算法的联系与区别?

相同点：

1. 两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
2. Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交
3. ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点：

ZAB(ZooKeeper Atomic Broadcast) 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

而 Paxos 算法与 ZAB 协议不同的是，Paxos 算法的发起者可以是一个或多个。当集群中的 Acceptor 服务器中的大多数可以执行会话请求后，提议者服务器只负责发送提交指令，事务的执行实际发生在 Acceptor 服务器。这与 ZooKeeper 服务器上事务的执行发生在 Leader 服务器上不同。Paxos 算法在数据同步阶段，是多台 Acceptor 服务器作为数据源同步给集群中的多台 Learner 服务器，而 ZooKeeper 则是单台 Leader 服务器作为数据源同步给集群中的其他角色服务器。

注意：

ZAB是在Paxos的基础上改进和演变过来的。

提议者（Proposer）、决策者（Acceptor）、决策学习者（Learner）

8. Zookeeper 的典型应用场景

1. 数据发布/订阅
2. 负载均衡
3. 命名服务
4. 分布式协调/通知
5. 集群管理

6. Master 选举
7. 分布式锁
8. 分布式队列

数据发布/订阅系统，即所谓的配置中心，目的：动态获取数据（配置信息），实现数据（配置信息）的集中式管理和数据的动态更新

Zookeeper 分布式锁

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode的方式来实现。所有客户端都去创建 /task_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 task_lock 节点就释放出锁。

对于第二类， /task_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

Zookeeper 队列管理

> 一般很少用到，可简单了解

两种类型的队列：

1. 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
2. 队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

9. Chroot特性

zookeeper v3.2.0 版本后，添加了 Chroot 特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置 Chroot，能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用共用一个 Zookeeper 进群的场景下，对实现不同应用间的相互隔离非常有帮助。

拓展

ZooKeeper以Fast Paxos算法为基础，Paxos 算法存在活锁的问题，即当有多个 proposer 交错提交时有可能互相排斥导致没有一个proposer能提交成功，而Fast Paxos做了一些优化，通过选举产生一个领导者，只有leader才能提交proposer具体算法可见Fast Paxos。

点击在线阅读《最少必要面试题》