# JAVA基础部分

## 1.1、基础内容【必须】

#### JDK 和 JRE 有什么区别？

JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。

JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。

**具体来说 JDK 其实包含了 JRE，同时还包含了编译 Java 源码的编译器 Javac，还包含了很多 Java 程序调试和分析的工具。**简单来说：如果你需要运行 Java 程序，只需安装 JRE 就可以了，如果你需要编写 Java 程序，需要安装 JDK。

#### 抽象类与接口区别

关键点：抽象类可以有构造方法，接口中不能有构造方法、抽象类中可以有普通成员变量，接口中没有普通成员变量、一个类可以实现多个接口，但只能继承一个抽象类。

#### == 和 equals 的区别是什么？

== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；

equals 默认情况下是引用比较，只是很多类重新了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

#### Stream常用方法：

* map: 用作类型转换 如把集合里面的字符串转为大写,或者一个对象的集合取几个字段转为新的对象集合
* filter: 过滤 符合条件的集合元素保存下来,不符合条件的去掉

#### final 在 Java 中有什么作用？

final 修饰的类叫最终类，该类不能被继承。  
 final 修饰的方法不能被重写。  
 final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

#### String 类的常用方法都有那些？

**indexOf()：返回指定字符的索引。**

charAt()：返回指定索引处的字符。

**replace()：字符串替换。**

trim()：去除字符串两端空白。

**split()：分割字符串，返回一个分割后的字符串数组。**

getBytes()：返回字符串的 byte 类型数组。

length()：返回字符串长度。

toLowerCase()：将字符串转成小写字母。

toUpperCase()：将字符串转成大写字符。

**substring()：截取字符串。**

equals()：字符串比较。

#### 字符串中replace和replaceAll的区别？

* replace的第一个参数是char和CharSequence,即可以支持字符和字符串的替换。
* replaceAll的第一个参数是regex，即基于规则表达式的替换。

#### String, StringBuilder,StringBuffer三者的区别

* String和StringBuilder的本质区别

String是一个不可改变的字符序列.

StringBuilder是一个可以改变的字符序列.

* 常见的字符拼接,该选择谁

推荐使用StringBuilder,因为拼接的效率高

* StringBuilder和StringBuffer的区别

StringBuffer是线程安全(同步)，但是效率低.

StringBuilde是线程不安全(不同步)，但是效率高.

#### JAVA反射机制提供了什么功能

**Java反射机制提供如下功能**：

* 在运行时判断任意一个对象所属的类
* 在运行时构造任意一个类的对象
* 在运行时判段任意一个类所具有的成员变量和方法
* 在运行时调用任一个对象的方法
* 在运行时创建新类对象
* 在使用Java的反射功能时，基本首先都要获取类的Class对象，再通过Class对象获取其他的对象。

Class 🡪 Contructor + Field + Method

#### 常见的异常类有哪些？

* NullPointerException 空指针异常【少做空判断】
* ClassNotFoundException 指定类不存在【包冲突】
* NumberFormatException 字符串转换为数字异常【1a23：检查：正则判断，hutool工具类】
* IndexOutOfBoundsException 数组下标越界异常
* ClassCastException 数据类型转换异常
* FileNotFoundException 文件未找到异常
* NoSuchMethodException 方法不存在异常
* IOException IO 异常
* SocketException Socket 异常

#### Java中异常处理机制

**体系结构**

Throwable

Error：严重性错误

Exception：

RuntimeException: 运行时异常

!RuntimeException: 编译时异常

* 异常的处理方式

总结：逐层上抛，在springmvc中使用统一异常处理来处理。

@ControllerAdvice + @ExceptinHandler(xxx.class)

#### Java中finally和return执行顺序

[(48条消息) Java中finally和return执行顺序\_liuhmmjj的博客-CSDN博客\_finally return执行顺序](https://blog.csdn.net/u014082714/article/details/106050735)

* finally语句在return语句执行之后 return返回之前执行的
* finally块中的return语句会覆盖try块中的return返回
* 如果finally语句中没有return语句覆盖返回值，那么原来的返回值可能因为finally里的修改而改变也可能不变
* try块里的return语句在异常的情况下不会被执行，这样具体返回哪个看情况
* 当发生异常后，catch中的return执行情况与未发生异常时try中return的执行情况完全一样

#### 常见的基本排序

总结：记住一两个常见的排序【冒泡】，自己百度去

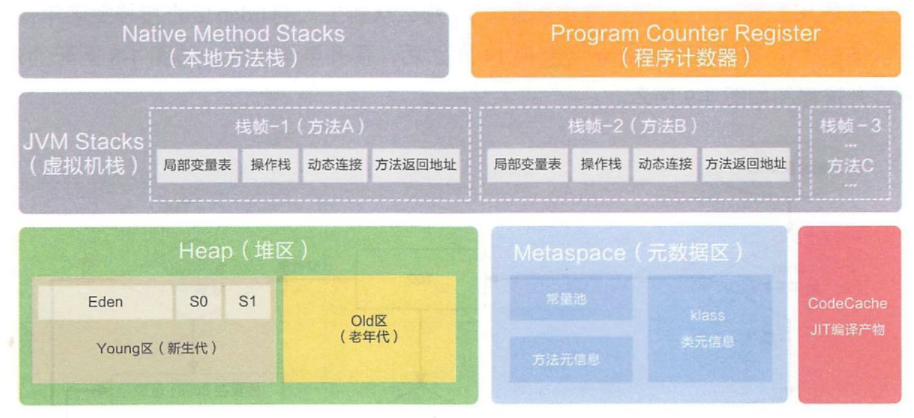
## 1.2、JVM虚拟机&类加载

#### Java 内存区域【必须，记住结果】

Java 虚拟机在执行 Java 程序的过程中会把他所管理的内存划分为若干个不同的数据区域。Java 虚拟机规范将 JVM 所管理的内存分为以下几个运行时数据区：

程序计数器、Java 虚拟机栈、本地方法栈、Java 堆、元数据区。

JDK8 之前，Hotspot 中方法区的实现是永久代（Perm），JDK8 开始使用元空间（Metaspace），以前永久代所有内容的字符串常量移至堆内存，其他内容移至元空间，元空间直接在本地内存分配。



#### JAVA中垃圾回收机制【记住常见的垃圾回收算法】

* 什么样的对象会被当做垃圾回收

当一个对象的引用(地址)没有变量去记录的时候,该对象就会成为垃圾对象,并在垃圾回收器空闲的时候对其进行清扫.

* 如何检验对象是否被回收

可以重写Object类中的finalize方法

这个方法在垃圾回收器执行的时候,被回收器自动调用执行的.

* 怎样通知垃圾回收器回收对象

可以调用system类的静态方法gc().通知垃圾回收器去清理垃圾.

1、标记清除算法

特点： 简单、收集速度快，但会有空间碎片，空间碎片会导致后面的GC频率增加。

适合场景：只有小部分对象需要进行回收的，适用于老年代的垃圾回收，因为老年代一般存活对象会比回收对象要多。

2、标记复制算法

特点：收集速度快，可以避免空间碎片，但是有空间浪费，存活对象较多的情况下复制对象的过程等会非常耗时，而且需要担保机制。

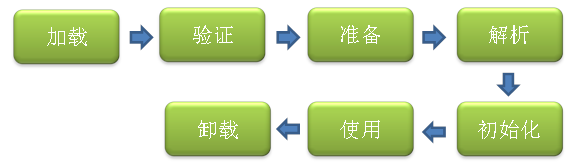
适合场景： 只有少量对象存活的场景，这也正是新生代对象的特点，所以一般新生代的垃圾回收器基本都会选择标记复制法。

3、标记整理算法

特点： 相对于标记复制法不会浪费内存空间，相对标记清除法则可以避免空间碎片，但是速度比其他两个算法慢。

适合场景： 内存吃紧，又要避免空间碎片的场景，老年代想要避免空间碎片问题的话通常会使用标记整理法。

#### 类加载过程【必须】

加载 -> 验证 -> 准备 -> 解析 -> 初始化：

* 加载:

1.获取类的二进制字节流

2.将字节流代表的静态存储结构转化为方法区运行时数据结构，在堆中生成class字节码对象

* 验证:连接过程的第一步,确保class文件的字节流中的信息符合当前虚拟机的要求,不会危害虚拟机的安全
* 准备:为类的静态变量分配内存并将其初始化为默认值
* 解析:虚拟机将常量池内符号引用替换成直接引用的过程
* 初始化:执行类构造器的init的过程

#### Java内存溢出

[(48条消息) 9.内存溢出\_zuishengmengsi1990的博客-CSDN博客](https://blog.csdn.net/zuishengmengsi1990/article/details/120093015)

* Java堆溢出：

Java堆用于存储对象实例，只要不断地创建对象，当对象数量到达最大堆的容量限制后就会产生内存溢出异常。最常见的内存溢出就是存在大的容器，而没法回收，比如：Map，List等。

java.lang.OutOfMemoryError: Java heap space

* 虚拟机栈和本地方法栈溢出

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常，请见情况：递归调用，无法退出。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常，线程创建过多导致，每个线程需要申请栈空间。

java.lang.OutOfMemoryError: unable to create new native thread

* 元数据区域的内存溢出：

java.lang.OutOfMemoryError: Metaspace

元数据区域或方法区是用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。我们可以通过在运行时产生大量的类去填满方法区，直到溢出，如：代理的使用(CGlib)、大量JSP或动态产生JSP文件的应用（JSP第一次运行时需要编译为Java的Servlet类）、基于OSGi的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等

* 运行时常量池的内存溢出：

String.intern()是一个Native方法，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。

java.lang.OutOfMemoryError : PermGen space

* 直接内存溢出

使用这个unsafe类的native方法allocateMemory去直接申请内存导致的内存溢出。

**总结：记住前两个！！**

#### java内存泄漏

[Java 内存泄漏原因、解决办法及泄漏排查 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/368830445)

## 1.4、多线程【必须】

#### 线程和进程的区别？

* 进程：进程指正在运行的程序。确切的来说，当一个程序进入内存运行，即变成一个进程，进程是处于运行过程中的程序，并且具有一定独立功能。
* 线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

**简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程**

#### 创建线程的方式

* 继承Thread类

优点:代码简单

缺点:该类无法继承别的类

* 实现Runnable接口

优点:继承其他类.统一实现该接口的实例可以共享资源.

缺点:代码复杂.

* 实现Callable接口

Callable中的call()方法有返回值,其他和Runnable的run()方法一样.

* 线程池方式

优点:实现自动化装配,易于管理,循环利用资源.

**Runnable和 Callable 的区别，Callabe可以有返回值。**

#### Java中wait方法和sleep方法的不同之处

**共同点：**

wait() ，wait(long) 和 sleep(long) 的效果都是让当前线程暂时放弃 CPU 的使用权，进入阻塞状态

**不同点：**

* 方法归属不同

sleep(long) 是 Thread 的静态方法，而 wait()，wait(long) 都是 Object 的成员方法，每个对象都有

* 醒来时机不同

执行 sleep(long) 和 wait(long) 的线程都会在等待相应毫秒后醒来

wait(long) 和 wait() 还可以被 notify 唤醒，wait() 如果不唤醒就一直等下去

它们都可以被打断唤醒

* 锁特性不同（重点）

wait 方法的调用必须先获取 wait 对象的锁，而 sleep 则无此限制；

wait 方法执行后会释放对象锁，允许其它线程获得该对象锁（我放弃 cpu，但你们还可以用）；

而 sleep 如果在 synchronized 代码块中执行，并不会释放对象锁（我放弃 cpu，你们也用不了）。

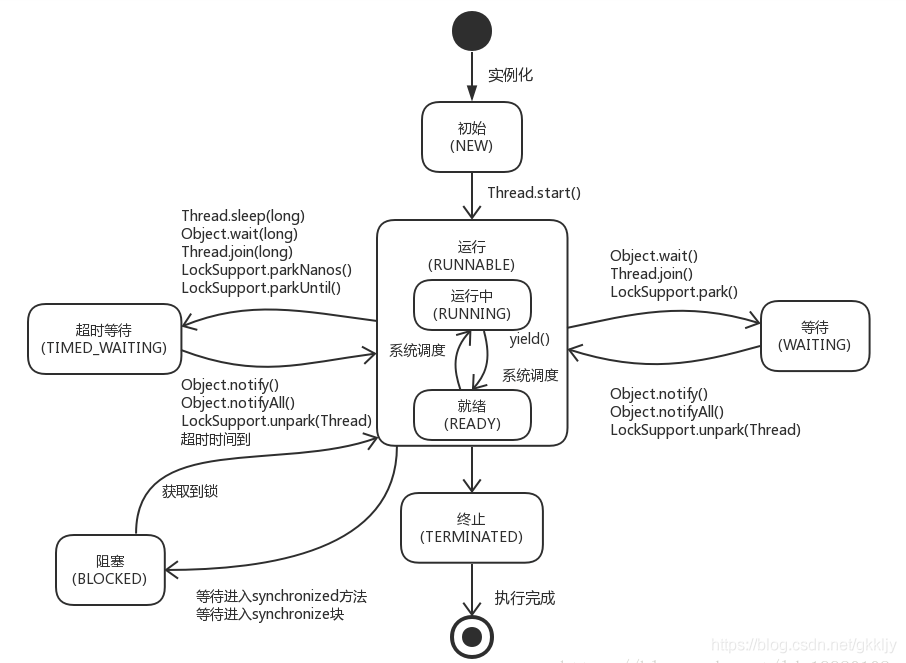
#### 线程的状态

* 就绪(Runnable):线程准备运行，不一定立马就能开始执行。
* 运行中(Running)：进程正在执行线程的代码。
* 等待中(Waiting):线程处于阻塞的状态，等待外部的处理结束。
* 睡眠中(Sleeping)：线程被强制睡眠。
* 阻塞：

I/O阻塞(Blocked on I/O)：等待I/O操作完成。

同步阻塞(Blocked on Synchronization)：等待获取锁。

* 死亡(Dead)：线程完成了执行。



**总结：**

start -> 运行 -> 结束

运行 -> wait() / join() 【等待】 -> notify()/notifyAll() -> 运行

运行 -> sleep(10)/wait(10)/join(10) 【超时等待】 -> notify()/notifyAll()/超时 -> 运行

运行 -> synchroinzied 争锁 【阻塞】 -> 获取锁 -> 运行

#### 线程池的种类【必须】

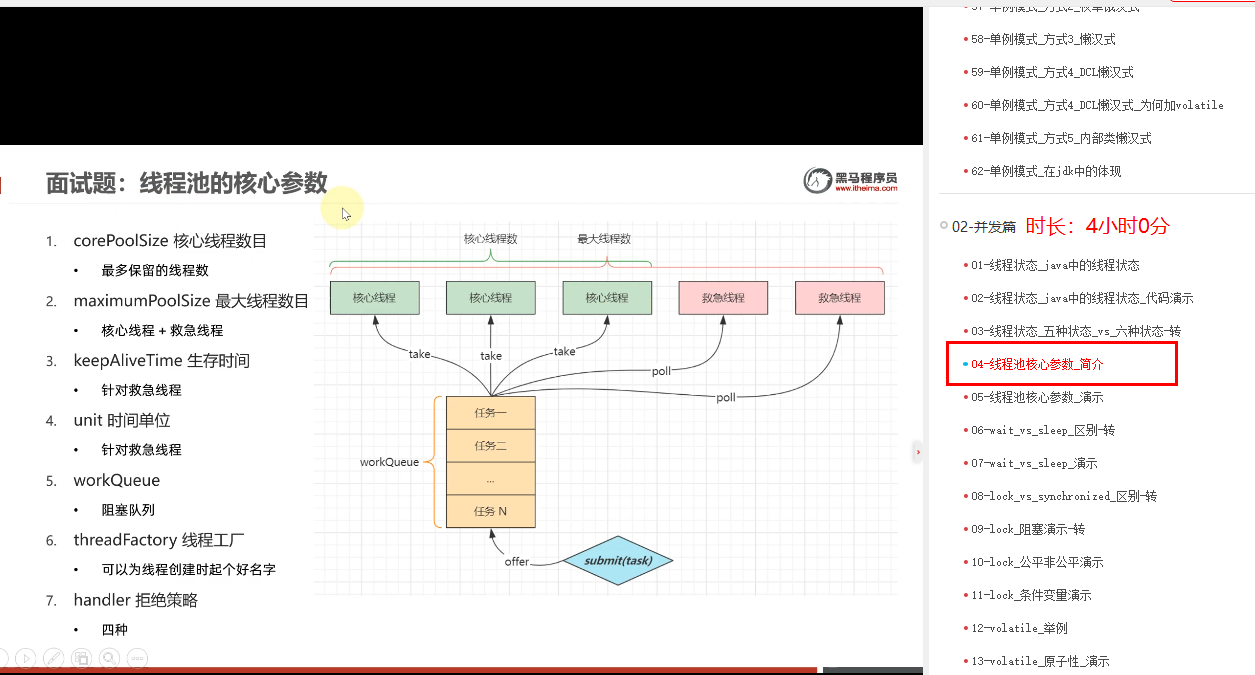
* newCachedThreadPool创建一个**可缓存线程池**，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
* newFixedThreadPool 创建一个**定长线程池**，可控制线程最大并发数，超出的线程会在队列中等待。
* newScheduledThreadPool 创建一个**定长线程池，支持定时及周期性任务执行**。
* newSingleThreadExecutor 创建一个**单线程化的线程池**，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行

因为在项目中，线程的创建和销毁非常消耗资源，所以使用在多线程场景的时候会使用线程池，根据自己业务逻辑的需求，使用不同的线程池。

**一个都不建议使用！！！！**

#### 线程池的核心参数？【必须】

自己找！！看tlias上的视频



#### volatile与synchronized的区别，底层实现

* 首先两者都是用来确保数据的一致性的，volatile它能够使变量在值发生改变时尽快让其他线程知道，为什么要这么做呢？编译器为了加快程序运行速度，对一些变量的写操作会现在寄存器或者是cpu缓存上进行，最后才写入内存，这个过程，变量的新值对其他线程是不可见的，而volatile的作用就是使它修饰的变量的读写操作都必须在内存中进行

区别：

* Volatile本质是告诉jvm当前变量在寄存器中的值是不安全的需要从内存中读取，sychronized则是锁定当前变量，只有当前线程可以访问到该变量其他线程被阻塞
* Volatile只能作用于变量，synchronized则是可以使用在变量和方法上
* Volatile仅能实现变量的修改可见性,但不具备原子特性,而synchronized则可以保证变量的修改可见性和原子性
* volatile不会造成线程的阻塞,而synchronized可能会造成线程的阻塞
* volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化

注意事项：

* 在使用volatile关键字时要慎重，并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原来操作，当变量的值由自身的上一个决定时，如n=n+1、n++ 等，volatile关键字将失效，只有当变量的值和自身上一个值无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile

#### 什么是死锁？【必须】

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

#### 怎么防止死锁？【必须】

* 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用 Java.util.concurrent 并发类代替自己手写锁。
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
* 尽量减少同步的代码块。

## 1.5、IO模型【3年左右】

#### IO对比总结

* IO 的方式通常分为几种：同步阻塞的 BIO、同步非阻塞的 NIO、异步非阻塞的 AIO。
* BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。
* NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。
* AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

举个例子：

* 同步阻塞：你到饭馆点餐，然后在那等着，啥都干不了，饭馆没做好，你就必须等着！
* 同步非阻塞：你在饭馆点完餐，就去玩儿了。不过玩一会儿，就回饭馆问一声：好了没 啊！
* 异步非阻塞：饭馆打电话说，我们知道您的位置，一会给你送过来，安心玩儿就可以了， 类似于现在的外卖。

[8. Java-IO模型（BIO、NIO、AIO） - 简书 (jianshu.com)](https://www.jianshu.com/p/670033e5b916)

## 1.6、Java容器【必须】

#### Java 容器都有哪些？

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

Collection：List【ArrayList】 Set【HashSet】

List、ArrayList、LinkedList、Vector、Stack、Set、HashSet、LinkedHashSet、TreeSet

Map：

HashMap、LinkedHashMap、TreeMap、ConcurrentHashMap、Hashtable

#### 数组与链表的区别？

**总结：**

**数组随机访问性强，查找速度快，但是插入和删除效率低；而且数组大小固定，不能动态扩展，数组要求必须有连续的内存空间，内存空间要求高。**

**链表插入删除速度快，大小不固定，拓展很灵活。但是不能随机查找，必须从第一个开始遍历，查找效率低**

#### Map集合几种遍历方式?

第一种: 通过Map.keySet获取key的Set集合，之后在通过key进行遍历

第二种: 通过Map.values获取所有value，之后再进行遍历

第三种: 通过Map.entrySet获取Set集合，之后通过iterator进行遍历

第四种: 直接通过foreach对Map.entrySet获取的Set集合进遍历

#### List集合的特性

有序可重复的集合

#### 如何实现数组和 List 之间的转换？

数组转 List：使用 Arrays.asList(array) 进行转换。

List 转数组：使用 List 自带的 toArray() 方法。

#### ArrayList 和 LinkedList 有什么区别

* **ArrayList是实现了基于动态数组的数据结构，LinkedList是基于链表结构。**
* 对于随机访问的get和set方法，ArrayList要优于LinkedList，因为LinkedList要移动指针。
* 对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

总结：抓住第一个问题！！

#### list、set和map的区别

**总结：**

**list和set都是实现了collection接口的集合，其中list是有序可重复的，而set是无须不可重复的；**

**而Map是一个独立的接口，不是collection的子接口，map中存储的是key-value键值对的数据。**

#### ArrayList的扩容规则？



#### 说一下 HashMap 的实现原理？【重中之重】

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。

#### 说一下 HashSet 的实现原理？

HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。

#### CurrentHashMap了解吗?

HashMap是线程不安全的 , 但是效率高 , HashTable是线程安全的 , 但是效率低.有没有一种对象是即是线程安全的 , 同时执行效率可以达到HashMap呢?

CurrentHashMap可以做到。底层实现通过分段加锁进行实现 , hashmap底层是数组加上链表实现的 , 那么一个线程来操作数据,只是操作数组中一个索引的数据. 如果此时对整个数组加锁,其他线程操作不了这个数组,所以效率低.其实线程也就操作数组的一个索引,对这个索引进行加锁 , 而锁对象就是这个索引所对应的值,其他线程来修改其他索引数据时,拿到的是其他索引的锁对象,从而提高了效率.

## 1.7、Java设计模式

#### 常见设计模式

* **工厂模式**：一个抽象接口的实现，多个抽象接口的实现类，spring的beanFactory就是工厂模式 [(48条消息) 工厂设计模式\_coyhzx的博客-CSDN博客\_工厂设计模式](https://blog.csdn.net/qq_39504520/article/details/106364731)
* **单例模式**：在内存中，保证对象的实例只有一个。
* 装饰者模式：对一个类进行装饰，增强其方法行为，如Java中的IO流就使用了装饰者模式
* **代理模式**：比如sping AOP使用动态代理
* 适配器模式：io流，通过继承实现将一个接口适配到另一个接口，InputStreamReader类继承Reader接口，但要创建它们必须在构造函数中传入一个InputStream的实例，InputStreamReader的作用也就是将InputStream适配到Reader
* 状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

#### 单例设计模式中懒汉式和饿汉式的区别

* 饿汉式：

//饿汉式单例类.在类初始化时，已经自行实例化

public class Singleton1 {

private Singleton1() {}

private static final Singleton1 single = new Singleton1();

//静态工厂方法

public static Singleton1 getInstance() {

return single;

}

}

* 懒汉式：

//懒汉式单例类.在第一次调用的时候实例化自己

public class Singleton {

private Singleton() {}

private static Singleton single=null;

//静态工厂方法

public static Singleton getInstance() {

if (single == null) {

single = new Singleton();

}

return single;

}

}

* 饿汉式就是类一旦加载，就把单例初始化完成，保证getInstance()的时候，单例就已经存在。
* 懒汉式比较懒，只有当调用getInstance的时候，才会去初始化这个单例

区别：

* 饿汉式是线程安全的，懒汉式是线程不安全的（即一个进程内有多个线程在在同时使用时可能会产生多个实例，可创建个静态内部类，产生一个单例对象，通过静态内部类返回获取这个对象）

[单例设计模式精讲 (baidu.com)](https://baijiahao.baidu.com/s?id=1712870063861265422&wfr=spider&for=pc)

**总结：能说出两三个以上的设计模式，重点掌握工厂和单例设计模式。**

# WEB

1. **cookie和session的区别与联系【必须】**
2. cookie数据存放在客户的浏览器上,session数据存放在服务器上.
3. 很多浏览器限制站点最多保存20个cookie,单个cookie保存的数据不能超过4k.
4. cookie不是很安全,考虑安全应当使用session.
5. 可以考虑将登录信息等重要信息存放为session,其它信息如果需要保留,可以放在cookie中.
6. session会在一定时间内保存在服务器上.
7. session会在浏览器关闭或者一段时间内销毁,也可以通过setMaxInactiveInterval(int)方法进行设置,或是通过invalidate()方法强制结束当前会话.cookie可以通过setMaxAge(int)方法设置缓存在客户端的时间.
8. 一般情况下,session生成的sessionid都是保存在cookie中.

**总结:**

**cookie:在客户端保存数据,不安全.只能保存字符串,且是少量数据.**

**session:在服务器端保存数据,安全.可以保存对象数据,数据无限制.**

Session是基于cookie的，session的id是存在cookie中的

1. **如果客户端禁止cookie，session 还能用吗？【必须】**

可以用，session 只是依赖 cookie 存储 sessionid，如果 cookie 被禁用了，可以使用 url 中添加 sessionid 的方式保证 session 能正常使用。url重写！！

1. **Get方法和Post方法区别【必须】**

get:查询、请求参数 不安全,提交的数据量小

post:添加或更新数据、提交数据比较安全、提交的数据量无限制.

1. get方法用于信息获取,他是安全的,而post方法是用于修改服务器上资源的请求.
2. get请求的数据会附在url之后,而post方法提交的数据则放置在http报文实体的主体里,所以post方法的安全性比get方法要高.
3. get方法传输的数据量一般限制在2kb,post方法对于数据大小是无限制的.
4. **常见的http返回状态码【必须】**

100:告诉客户端应继续发送请求

200:请求响应成功

202:请求已被受理还未做出响应

301：永久重定向

302：暂时重定向

400:请求无效,常见的情况是请求参数有误,http头构建错误

404:访问不到资源

500:服务器后端错误

1开头的状态码是消息类型的.

2开头的状态码表示成功.

3开头的状态码表示需要重定向.

4开头的状态码表示请求错误.

5开头的状态码表示服务器错误.

1. **TCP和UDP的区别,HTTP协议【必须】**

* TCP协议提供安全可靠的网络传输服务,它是一种面向连接的服务.类似于打电话,必须先拨号.双方建立一个传递信息的通道传输.
* UDP协议是一种数据报协议,它传输的数据是分组报文,它是无连接的,不需要和目标通信方建立连接,类似于写信,所以它的传输不保证安全可靠.但适合大数据量的传输.
* HTTP协议是超文本传输协议,是一种相对于TCP来说更细致的协议,TCP以及UDP协议规范的是网络设备之间的通信规范,HTTP实在TCP协议的基础上针对用户的协议,用户服务具体体现在应用程序之间的交互,比如javaweb中客户端服务端体系就要用http协议来规范通信.

TCP和UDP在开发中很少见到,但是网络底层都有他们的影子,正常的会话级别的服务:如客户端服务器体系底层就说基于TCP协议.而邮件发送,短信发送等底层使用的是UDP协议.

HTTP协议,客户端/服务器体系的程序都使用HTTP协议来规范通信.

1. **tcp为什么要三次握手，两次不行吗？为什么？**

如果采用两次握手，那么只要服务器发出确认数据包就会建立连接，但由于客户端此时并未响应服务器端的请求，那此时服务器端就会一直在等待客户端，这样服务器端就白白浪费了一定的资源。若采用三次握手，服务器端没有收到来自客户端的再此确认，则就会知道客户端并没有要求建立请求，就不会浪费服务器的资源。

1. **如何实现跨域？【必须】**
   * 服务器端运行跨域 设置 CORS 等于 \*；
   * 在单个接口使用注解 @CrossOrigin 运行跨域；

* 使用 jsonp 跨域,JSONP 实现原理:

jsonp：JSON with Padding，它是利用script标签的 src 连接可以访问不同源的特性，加载远程返回的“JS 函数”来执行的。

总结：站在后端角度出发，在微服务角度来说，跨域需要在网关层面解决，只需要配置即可。

使用的是CORS解决方案！！

# 数据库【必须】

1. **事务的特性和隔离级别**

事务的特性：

* 原子性（Atomicity）

原子性指事务是不可分割的工作单位，事务中的操作要么都发生，要么都不发生

* 一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态

* 隔离性（Isolation）

事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

* 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

隔离级别：

* 未提交读read uncommitted

会发生 脏读、不可重复读、虚读

* 已提交读read committed //Oracle SQL Server（系统事务）

解决脏读，但是不可重复读和虚读有可能发生

* 重复读repeatable read //Mysql

解决脏读和不可重复读，但是虚读有可能发生.

* 串行化serializable

避免脏读，不可重复读，虚读的发生

不考虑隔离性会产生的问题：

* 脏读：脏读是读到了别的事务回滚前的脏数据。比如事务B执行过程中修改了数据X，在未提交前，事务A读取了X，而事务B却回滚了，这样事务A就形成了脏读。
* 不可重复读：事务A首先读取了一条数据，然后执行逻辑的时候，事务B将这条数据改变了，然后事务A再次读取的时候，发现数据不匹配了，就是所谓的不可重复读了
* 幻读：事务A首先根据条件索引得到N条数据，然后事务B改变了这N条数据之外的M条或者增添了M条符合事务A搜索条件的数据，导致事务A再次搜索发现有N+M条数据了，就产生了幻读。

1. **Mysql的体系结构**

* 连接层

最上层是一些客户端和链接服务，包含本地sock 通信和大多数基于客户端/服务端工具实现的类似于 TCP/IP的通 信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证 它所具有的操作权限。

* 服务层

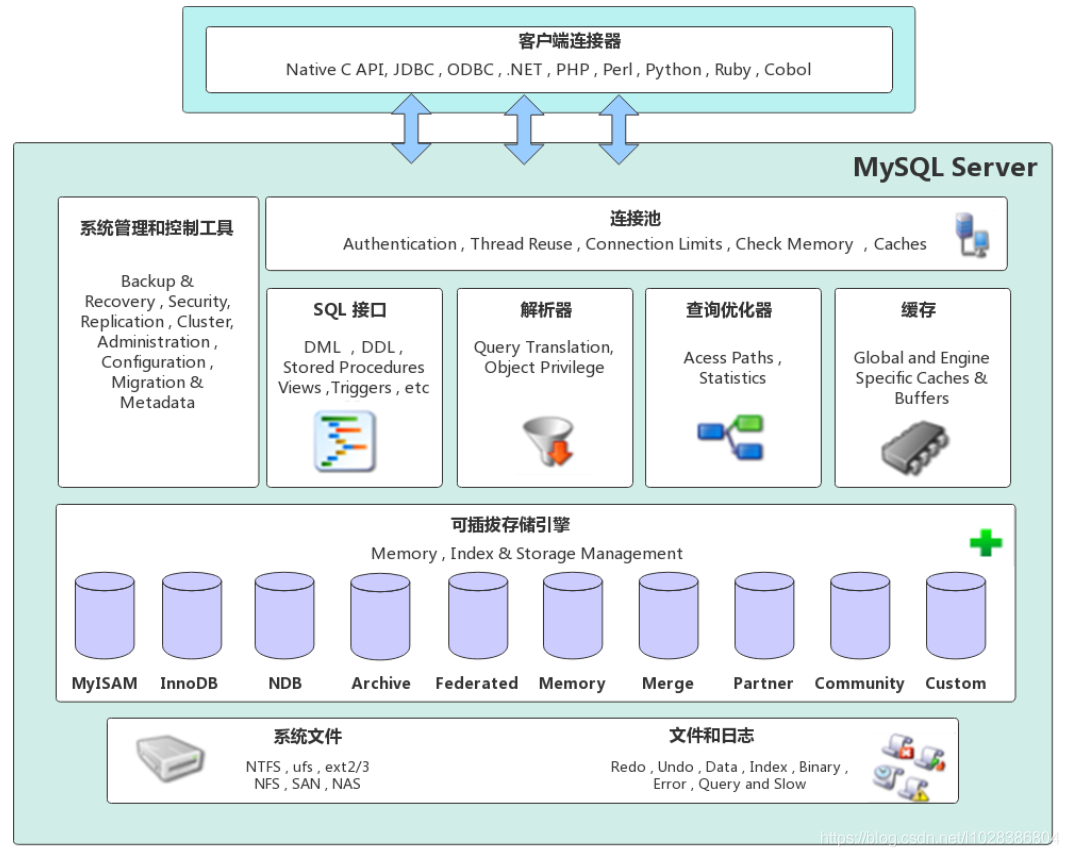
第二层架构主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部分内置函数的执行。所有跨存储引擎的功能也在这一层实现，如 过程、函数等。在该层，服务器会解析查询并创建相应的内部 解析树，并对其完成相应的优化如确定表的查询的顺序，是否利用索引等， 最后生成相应的执行操作。如果是 select语句，服务器还会查询内部的缓存，如果缓存空间足够大，这样在解决大量读操作的环境中能够很好的提升系统的性能。

* 引擎层

存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，

* 存储层 服务器通过API和存储引擎进行通信。不同的存储引擎具有不同的功能，这样我们可以根据自己的需要，来选取合适的存储引擎。

数据存储层，主要是将数据存储在文件系统之上，并完成与存储引擎的交互。和其他数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎上，插件式的存储引擎架构，将查询处理和其他的系统任务以及数据的存储提取分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。



1. **InnoDB，MyISAM存储引擎特性**

* InnoDB

InnoDB存储引擎是Mysql的默认存储引擎。InnoDB存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全。 但是对比MyISAM的存储引擎，InnoDB写的处理效率差一些，并且会占用更多的磁盘空间以保留数据和索引。

* MyISAM

MyISAM 不支持事务、也不支持外键，其优势是访问的速度快，对事务的完整性没有要求或者以SELECT、INSERT 为主的应用基本上都可以使用这个引擎来创建表 。

1. **CHAR和VARCHAR的区别：**

* CHAR和VARCHAR类型在存储和检索方面有所不同
* CHAR列长度固定为创建表时声明的长度，长度值范围是1到255
* 当CHAR值被存储时，它们被用空格填充到特定长度，检索CHAR值时需删除尾随空格。

1. **delete、drop、truncate区别**

* truncate 和 delete只删除数据，不删除表结构 ,drop删除表结构，并且释放所占的空间。
* 删除数据的速度，drop> truncate > delete
* delete属于DML语言，需要事务管理，commit之后才能生效。drop和truncate属于DDL语言，操作立刻生效，不可回滚。
* 使用场合：
* 当你不再需要该表时， 用 drop;
* 当你仍要保留该表，但要删除所有记录时， 用 truncate;
* 当你要删除部分记录时（always with a where clause), 用 delete.

1. **mysql 中 in 和 exists 区别**

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。  
 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in：  
not in 和not exists如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。  
 1.EXISTS只返回TRUE或FALSE，不会返回UNKNOWN。

2.IN当遇到包含NULL的情况，那么就会返回UNKNOWN。

1. **数据库的三范式是什么？【了解】**

* 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。
* 第二范式：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。
* 第三范式：任何非主属性不依赖于其它非主属性。

1. **Sql优化**

**总体方针：**

* 通过慢查询日志去寻找，哪些sql执行效率低。Mysql会将那些查询慢【时间阈值】的sql记录下来。
* 使用explain分析低效率的sql执行计划
* 针对低效率的sql执行计划分析

没有索引

索引失效

数据量太大



**情形一、没有索引：**

* 针对查询的列创建索引，提高查询效率，但是索引太多，mysql也会出现选择困难，所以建立索引要有效，无效的索引需要删除。

策略：

* 针对那些经常用于where条件查询的字段构建索引，针对进程排序的字段也可以构建索引。
* 尽量选择较小的列
* 将where中用的比较频繁的字段建立索引

**情形二、索引失效：**

* select子句中避免使用‘\*’
* 避免在索引列上使用计算，not，in和<>等操作
* 当只需要一行数据的时候使用limit 1
* 保证表单数据不超过200w，适时分割表
* 针对查询较慢的语句，可以使用explain来分析该语句具体的执行情况
* 避免查询时判断null，否则可能会导致全表扫描，无法使用索引;
* 避免like查询，否则可能导致全表扫描，可以考虑使用全文索引
* 能用union all的时候就不用union，union过滤重复数据要耗费更多的CPU资源

**情形三：数据量太大：**

分库分表：

* 使用MyCat中间件实现。

全文索引技术：

* ElasticSearch ， solr

非关系型数据库：

* 不需要像关系型数据库一样维护表于表之间的关系，而是使用json这种灵活多变的形式，效率比Mysql提高很多

1. **MySQL 索引是怎么实现的？**

索引是满足某种特定查找算法的数据结构，而这些数据结构会以某种方式指向数据，从而实现高效查找数据。

具体来说 MySQL 中的索引，不同的数据引擎实现有所不同，但目前主流的数据库引擎的索引都是 B+ 树实现的，B+ 树的搜索效率，可以到达二分法的性能，找到数据区域之后就找到了完整的数据结构了，所有索引的性能也是更好的。

1. **Mysql支持的索引类型**

* index  ----  普通索引,数据可以重复，没有任何限制。
* unique   ---- 唯一索引,要求索引列的值必须唯一，但允许有空值；如果是组合索引，那么列值的组合必须唯一。
* primary key ---- 主键索引,是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值，一般是在创建表的同时创建主键索引。
* 组合索引 ----  在多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。
* fulltext ---- 全文索引,是对于大表的文本域：char，varchar，text列才能创建全文索引，主要用于查找文本中的关键字，并不是直接与索引中的值进行比较。

1. **索引失效问题**

7种引起索引失效的情境

* 如果条件中有or，即使其中有部分条件带索引也不会使用(这也是为什么尽量少用or的原因)，例子中user\_id无索引。注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引
* 对于复合索引，如果不使用前列，后续列也将无法使用，类电话簿。
* like查询是以%开头
* 存在索引列的数据类型隐形转换，则用不上索引，比如列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引
* where 子句里对索引列上有数学运算，用不上索引
* where 子句里对有索引列使用函数，用不上索引
* 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

1. **左连接 ，右连接，内连接和全外连接的4者区别**

* **left join （左连接）**：返回包括左表中的所有记录和右表中连接字段相等的记录。
* right join （右连接）：返回包括右表中的所有记录和左表中连接字段相等的记录。
* **inner join （等值连接或者叫内连接）**：只返回两个表中连接字段相等的行。
* full join （全外连接）：返回左右表中所有的记录和左右表中连接字段相等的记录[mysql不支持]。

1. **说一下 MySQL 的行锁和表锁？【非必须】**

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。  
 行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

1. **说一下乐观锁和悲观锁？【非必须】**

乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。

悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻止，直到这个锁被释放。

数据库的乐观锁需要自己实现，在表里面添加一个 version 字段，每次修改成功值加 1，这样每次修改的时候先对比一下，自己拥有的 version 和数据库现在的 version 是否一致，如果不一致就不修改，这样就实现了乐观锁。

1. **聚集索引和非聚集索引的区别**

聚簇索引就是索引和记录紧密在一起。  
 非聚簇索引 索引文件和数据文件分开存放，索引文件的叶子页只保存了主键值，要定位记录还要去查找相应的数据块。

[(48条消息) 聚簇索引和非聚簇索引的区别\_xishilife的博客-CSDN博客\_聚簇索引和非聚簇索引的区别](https://blog.csdn.net/xishilife/article/details/120256375)

1. **mysql数据碎片处理**

在长期的数据更改过程中, 索引文件和数据文件,都将产生空洞,形成碎片.我们可以通过一个nop操作(不产生对数据实质影响的操作), 来修改表.比如: 表的引擎为innodb , 可以 alter table xxx engine innodb。也可以使用optimize table 表名 来进行修复.

注意: 修复表的数据及索引碎片,就会把所有的数据文件重新整理一遍,使之对齐.这个过程,如果表的行数比较大,也是非常耗费资源的操作.所以不能频繁的修复.如果表的Update操作很频率,可以按周/月,来修复.

如果不频繁,可以更长的周期来做修复.

# 框架

## 4.1、Spring

#### Spring 有哪些主要模块？

* spring core：框架的最基础部分，提供 ioc 和依赖注入特性。
* spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
* spring dao：Data Access Object 提供了JDBC的抽象层。
* spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
* spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
* spring Web mvc：spring 中的 mvc 封装包提供了 Web 应用的 Model-View-Controller（MVC）的实现。



#### IOC和DI【重点】

* IOC(控制反转).将对在自身对象中的一个内置对象的控制反转,反转后不再由自己本身的对象进行控制这个内置对象的创建,而是由第三方系统去控制这个内置对象的创建.简单来说就是把本来在类内部控制的对象,反转到类外部进行创建后注入,不再由类本身进行控制,这就是IOC本质. 对象交由spring创建和管理！
* DI(依赖注入).自身对象中的内置对象是通过注入的方式进行创建.
* DI是如何实现的.

依赖注入可以通过setter方法注入、构造注入、接口注入三种方法来实现,Spring支持setter注入和构造器注入、@Autowired自动注入,通常使用构造器注入来注入必须的依赖关系,对于可选的依赖关系,则setter注入是更好的选择,setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象.

#### Spring AOP【重点】

Spring AOP的面向切面编程【它可以在不改动原来代码的基础上对方法进行增强】,是面向对象编程的一种补充,用于处理系统中分布的各个模块的横切关注点,比如事务管理、日志、缓存、统一异常处理等.它是使用动态代理实现的,在内存中临时为增强某个方法生成一个AOP对象,这个对象包含目标对象的所有方法,在特定的切入点做了增强处理,并回调原来的方法.

Spring AOP的动态代理主要有两种方式实现,JDK动态代理和cglib动态代理.**JDK动态代理**通过反射来接受被代理的类,但是被代理的类必须**实现接口**,核心是InvacationHandler和Proxy类.cglib动态代理的类一般是没有实现接口的类,cglib是一个代码生成的类库,可以在运行时动态生成某个类的子类.所以,**cglib是通过继承的方式**做的动态代理,因此如果某个类被标记为final,那么它是无法使用cglib做动态代理的.

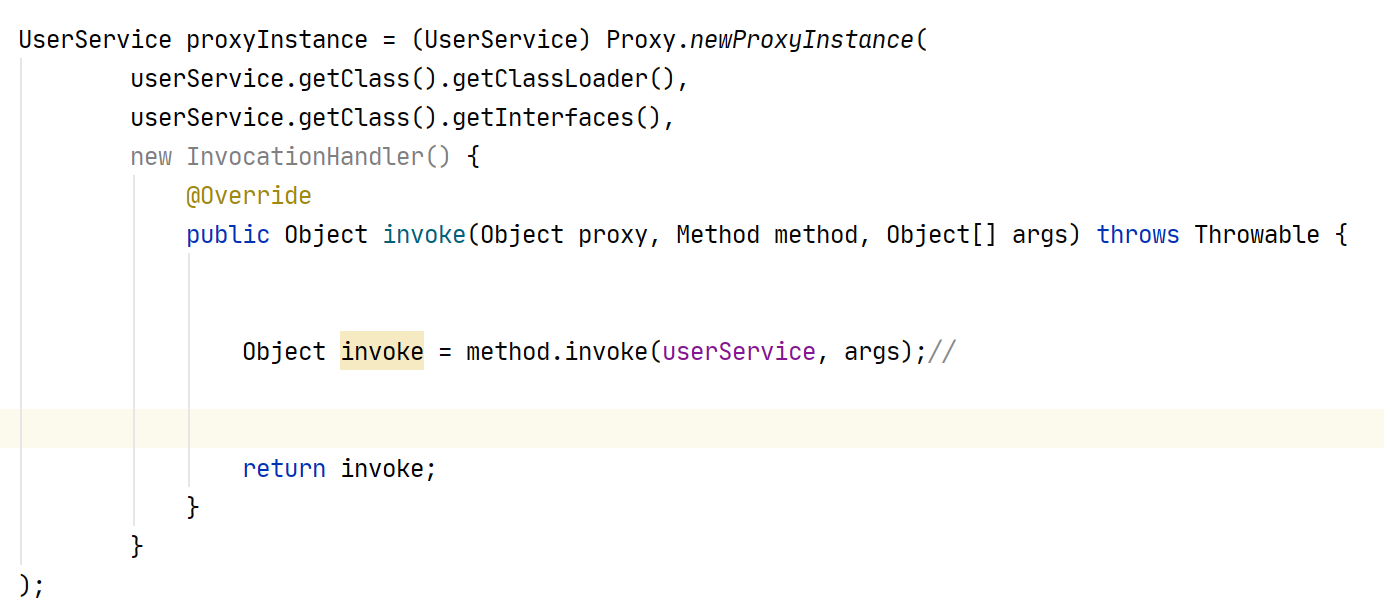
AOP能做什么:

* 降低模块的耦合度.
* 使系统容易扩展
* 避免修改业务代码,避免引入重复代码,更好的代码复用.

AOP怎么用:

* 前置通知:某方法调用前发出通知.
* 后置通知:某方法完成之后发出通知.
* 返回后通知:方法正常返回后,调用通知.在方法,正常退出发出通知.
* 异常通知:抛出异常后通知:在方法抛出异常退出时执行的通知.在方法调用时,异常退出发出通知.
* 环绕通知:通知包裹在被通知的方法的周围.

JDK动态代理：基于接口实现



Cglib：基于extends继承



#### Spring中Bean的作用域

Spring IOC容器在根据配置创建一个Bean对象实例时,可以为Bean指定实例的作用范围.

* **singleton(单例模式)**

IOC容器仅创建一个Bean实例,IOC容器每次返回的是同一个Bean实例.

* **prototype(原型模式)**

IOC容器仅创建多个Bean实例,IOC容器每次返回的是一个新的实例.

* request(HTTP请求)

该属性仅对HTTP请求产生作用,每次HTTP请求都会创建一个新的Bean,适用于WebApplicationContext.

* session(会话)

该属性仅用于HTTP Session,同一个session共享一个Bean实例.不同session使用不同的实例.

* global-session(全局会话,在spring5.x中已移除)

该属性仅用于HTTP Session,同session作用域不同的时候,所有session共享一个Bean实例.

总结：基本使用的都是singletong，而且spring默认就是singleton。面试官可能会说这个singleton会不会存在并发线程安全问题，问题是存在的。但是在程序中是不存在这个类问题的。

#### Spring框架实现实例化和依赖注入的方式

* 实例化:

1. 构造器实例化Bean
2. 静态工厂方式实例化Bean
3. 实例工厂方式实例化Bean

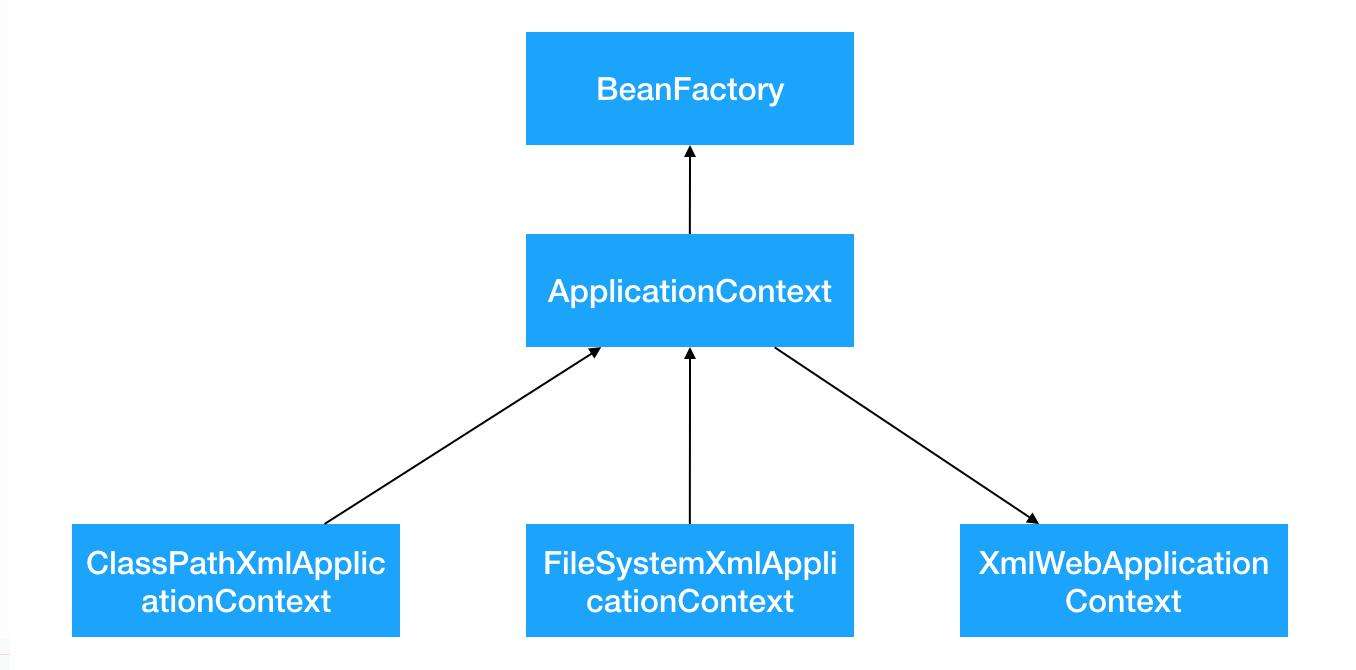
* 依赖注入:

1. 基于构造函数的注入
2. 基于set方法的注入
3. 基于自动装配的注入
4. 基于注解的依赖注入

#### Spring常用注解

* @Component:用于标记在一个类上,表示当前类是spring的一个组件,是ioc的一个容器.他有三个衍生注解:@Controller、@Service、@Repository
* @Controller：用于标记在一个类上,代表这个类是控制层组件.
* @Service:用于标记在一个类上,代表这个类是业务层组件.
* @Repository:用于标记在一个类上,代表这个类是数据访问层组件.、
* @Bean
* @Autowired
* @Value
* @Transactional:写在类上用于指定当前类中的方法支持事务,写在方法上表示当前的方法支持事务

#### BeanFactory类图结构



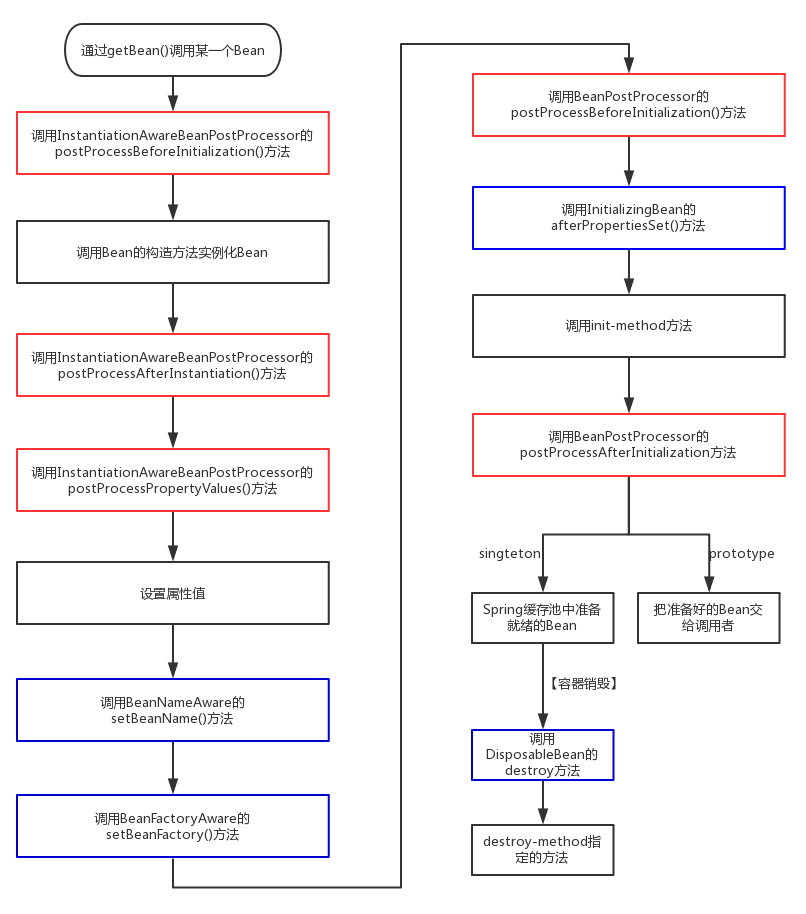
**BeanFactory VS ApplicationContext**

**所有的Bean在启动的时候都进行了加载 ApplicationContext**

**Bean在使用的时候才会加载 BeanFactory**

**BeanFactory【容器】 VS FactoryBean【创建bean放入容器】**

#### SpringBean的生命周期





**总结：创建的时候执行初始化方法 init-method【@PostConstruct】，在容器销毁的时候，执行销毁方法destroy-method【@PreDestroy】**

#### Spring 怎么解决循环依赖问题？【重要】

spring对循环依赖的处理有三种情况：

* + 构造器的循环依赖：这种依赖spring是处理不了的，直 接抛出BeanCurrentlylnCreationException异常。
  + 单例模式下的setter循环依赖：通过“三级缓存”处理循环依赖。
  + 非单例循环依赖：无法处理。

#### Spring 框架中用到了哪些设计模式？

* 工厂设计模式 : Spring使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
* 代理设计模式 : Spring AOP 功能的实现。
* 单例设计模式 : Spring 中的 Bean 默认都是单例的。

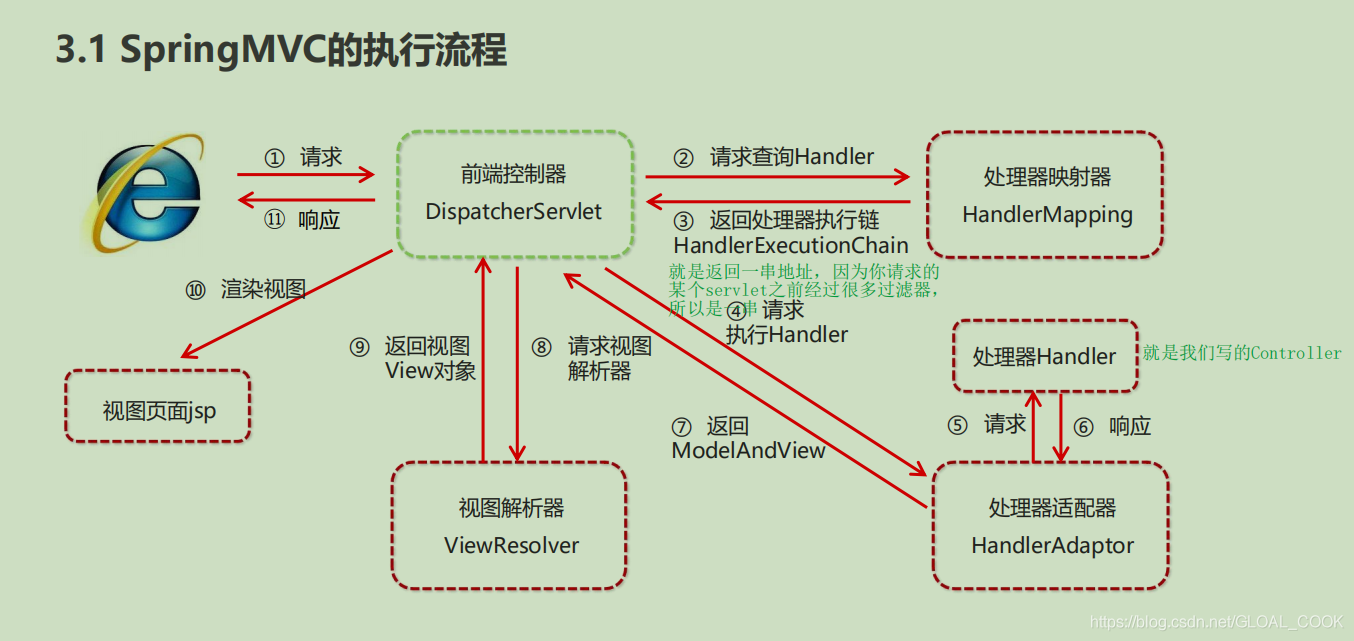
#### Spring事务传播行为

* **PROPAGATION\_REQUIRED: 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。**
* PROPAGATION\_SUPPORTS: 支持当前事务，如果当前没有事务，就以非事务方式执行。
* PROPAGATION\_MANDATORY: 支持当前事务，如果当前没有事务，就抛出异常。
* **PROPAGATION\_REQUIRES\_NEW: 新建事务，如果当前存在事务，把当前事务挂起。**
* PROPAGATION\_NOT\_SUPPORTED: 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* PROPAGATION\_NEVER: 以非事务方式执行，如果当前存在事务，则抛出异常。
* 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，则进行与PROPAGATION\_REQUIRED类似的操作。

## 4.2、SpringMVC

#### SpringMVC执行流程

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. HandlerAdapter执行处理器(handler，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View对象
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户



#### SpringMVC常用注解

* @RequestMapping:是一个用于处理请求地址映射的注解,可用于类或方法上.用于类上,表示类中所有响应请求的方法都是以该地址作为父路径
* @RequestParam:用于将指定的请求参数赋给方法中的形参.
* @PathVariable:可以获取URL中的动态参数.
* @RequestBody:用于读取request请求的body部分数据.
* @ResponseBody:用于将controller方法返回的对象,用流响应给客户端.
* @RestController:@Controller+@ResponseBody,用于标记在一个类上.

#### springmvc获取参数的几种方式

* 借助原始ServletAPI的HttpServletRequest对象.

Controller的方法,

添加HttpServletRequest类型入参,

通过HttpServletRequest.getParameter()获取请求数据

* 借助控制器方法的形参

controller的方法,添加参数来接收表单的数据.

接收的类型包括:

1. 基本类型或String
2. 实体类类型
3. 实体类关联对象

* Controller方法,接收实体类类型参数,而表单提交的是json数据时,可以使用@RequestBody注解获取全部请求体,配合jackson开源组件可以实现转换成实体类.要求json数据的key必须和实体类属性保持一致.

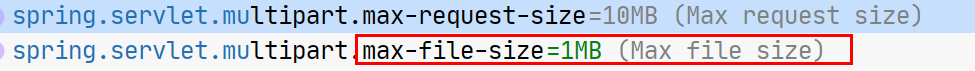
#### Spring MVC的异常处理 ？

* + - 编写异常处理类@ControllerAdvice/@RestControllerAdvice + 在类中编写异常处理方法@ExceptionHandler(Exception.class)
    - 编写异常处理类实现接口 HandlerExceptionResolver

#### Springmvc文件上传？

MultipartFile 接收文件 -> FileUpload提供api去做的 -> request的请求体

设置文件大小：



#### SpringMVC拦截器？

HandlerInterceptor 接口！！

## 4.3、Springboot

#### spring boot 有哪些方式可以实现热部署？

使用 devtools 启动热部署，添加 devtools 库，在配置文件中把 spring. devtools. restart. enabled 设置为 true；  
 使用 Intellij Idea 编辑器，勾上自动编译或手动重新编译。

#### SpringBoot 的常用注解有哪些？

@SpringBootApplication:

包含@Configuration、@EnableAutoConfiguration、@ComponentScan通常用在主类上；

@ComponentScan:组件扫描。个人理解相当于，如果扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean\*；  
@Configuration:指出该类是 Bean 配置的信息源，相当于XML中的，一般加在主类上；

@Bean:相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理；

@Import：用来导入其他配置类。

@ImportResource：用来加载xml配置文件。

@ControllerAdvice：包含@Component。可以被扫描到。统一处理异常。

@ExceptionHandler（Exception.class）：用在方法上面表示遇到这个异常就执行以下方法。

#### SpringBoot 有哪几种读取配置的方式？

方式一： 通过@Value("${spring.datasource.url}")这样的方式读取  
 方式二：通过@ConfigurationProperties(prefix = "spring.datasource")这种写法系统会依据prefix前缀自动注入配置数据到数据实体变量,这种方式不错,但是存在缺陷,我们编写的PropertyPlaceholderConfigurer扩展字段会无效,所以如果只是单纯的读取配置而不需要额外操作时可使用这方式最简单

方式三： 我们可以直接注入Environment对象示例并读取properties对象属性environment.getProperty("spring.datasource.database");与方式一的本质差不多,我们不需要编写对应字段的模型对象,但是对于程序可阅读性不好友,复用率不高

方式四： 通过系统启动时候初始化Listener,使用PropertiesLoaderUtils工具类读取指定配置文件并获得Properties配置对象,我们可以随时随地使用该对象的属性,这种方式比较少用,针对比较自定义的配置数据可使用该方式

#### SpringBoot 配置加载顺序？

在不指定要被加载文件时，默认的加载顺序：由里向外加载，所以最外层的最后被加载，会覆盖里层的属性，加载顺序依次为：

* 位于与jar包同级目录下的config文件夹，
* 位于与jar包同级目录下
* idea 环境下，resource文件夹下的config文件夹
* idea 环境下，resource文件夹下  （1->4, 外->里）

#### Spring Boot 如何定义多套不同环境配置

一、Spring Boot 环境设置机制

spring.profiles.active 属性可以为我们指定当前设置的环境，以此来选择我们的配置文件。例如我们有配置文件

application.yml

application-dev.yml

application-test.yml

application-prod.yml

当执行 java -jar xxx.jar --spring.profiles.actvie=test 此时，系统将启用 application.yml 和 application-test.yml配置文件。

当执行 java -jar xxx.jar --spring.profiles.actvie=prod 此时，系统将启用 application.yml 和 application-prod.yml 配置文件。

二、配置多环境

正如 第一 点所述，我们配置不同的配置文件

application.yml

application-dev.yml（开发环境）

application-test.yml（测试环境）

application-uat.yml（预发布环境）

application-prod.yml（生产环境）

三、指定环境

1 在 cmd 命令中指定

java -jar xxx.jar --spring.profiles.actvie=dev

2 在 application.yml 中指定

spring: profiles: active: dev

## 4.4、Mybatis&Mybatis-plus

#### Mybatis中使用#和$书写占位符有什么区别

#{}传参能防止sql注入.

${}传参是字符串拼接.

#### 动态SQL

* 所谓SQL的动态和静态,是指SQL语句在何时被编译和执行,二者都是用在SQL嵌入式编程中的.
* SQL语句的主体结构,在编译时尚无法确定,只有等到程序运行起来,在执行的过程中才能确定,这种SQL叫做动态SQL.
* 静态SQL语句的编译是在应用程序运行前进行的,编译的结果会存储在数据库内部.
* 程序运行时,数据库将直接执行编译好的SQL语句,降低运行时的开销.
* MyBatis中用于实现动态SQL的元素主要有:if、where、foreach.

#### Mapper动态代理规范

* xml映射文件中的namespace与mapper接口的全类名相同.
* mapper接口方法名和xml映射文件中定义的每个statement的id相同.
* mapper接口方法的输入参数类型和xml映射文件中定义的每个sql的parameterType的类型相同.
* mapper接口方法的输出参数类型和xml映射文件中定义的每个sql的resultType的类型相同.
* mybatis中的mapper动态代理是不支持方法重载的dao接口里的方法,因为是全类名+方法名的保存和寻找策略.
* mapper接口的工作原理是JDK动态代理,mybatis运行时就是用JDK动态代理为mapper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mappedStatement所代表的sql,然后将sql执行结果返回.

#### Mybatis常用注解

@Insert ： 插入sql , 和xml insert sql语法完全一样

@Select ： 查询sql, 和xml select sql语法完全一样

@Update ： 更新sql, 和xml update sql语法完全一样

@Delete ： 删除sql, 和xml delete sql语法完全一样

@Param ： 当映射器方法需要多个参数时，指定参数的名字

@Results ： 结果集合

@Result ： 结果

@One ：复杂类型的单独属性值映射

@Many ：复杂类型的己合属性映射

# Redis

1. **使用redis缓存的好处**

* 提高并发量，不需要每次都访问数据库，提高并发量。
* 提高性能，redis运行在内存中，内存的执行效率，远远超过数据库。

1. **Redis的key和value的存储大小有限制吗？**

* Redis的key和value的存储大小都是有限制的，都是512M。

1. **redis存储什么数据类型**
2. 字符串（String）
3. 字符串列表（lists）
4. 字符串集合（sets）
5. 有序字符串集合（sorted sets）
6. 哈希（hash）
7. **使用redis缓存的弊端【搁置】**

[(44条消息) Redis 缓存穿透、缓存击穿、缓存雪崩\_浩骞的博客-CSDN博客\_redis缓存穿透](https://blog.csdn.net/QiuHaoqian/article/details/109154315)

缓存穿透，大量的请求访问,查询的是数据库中不存在的数据,就会在缓存中无法命中,直接去访问数据库,数据库中也查询不到,自然无法将结果写入缓存,下一秒又有大量请求查询不存在的数据,导致数据库最终挂掉(要么系统出bug,要么黑客恶意攻击)

解决方案:

* 在数据库中没有查询到的数据,存储一个特定值到缓存中,这样下次恶意访问就可以去缓存中查询到数据,不会访问到数据库.注意的是这些特定的key,需要设置过期时间，避免黑客攻击时，大量的无效key把redis存满。

缓存雪崩，大规模的key失效，大量的请求通过key访问到数据库，导致数据库直接崩溃,然后系统直接瘫痪.

解决方案:

* 事前:保证redis集群的高可用性,redis cluster,主从机制;
* 事中:hystrix（熔断器）限流+降级,避免数据库被打死.高并发的访问量走hystrix限流组件,让访问量保持在数据库最大负载范围内,保证数据库不被打死,系统可以继续使用.剩余的请求走降级组件,返回一些默认的值或者是友情提示等
* 事后:redis持久化机制,尽快回复缓存集群,一旦重启,自动从磁盘上加载数据,回复内存中的数据

缓存击穿：

缓存与数据库不一致，修改数据时，数据库和缓存数据不一致。

最初级的缓存不一致:

出现场景：

* 先修改数据库,再删除缓存,如果修改缓存失败,就会导致数据库中式新数据,缓存中是旧数据,数据不一致

解决思路:

* 先删除缓存,再修改数据库,如果删除缓存成功,修改数据库失败,那么数据库中是旧数据,缓存中是空的,读的时候缓存中没有,去数据库中读旧数据,然后更新到缓存中去

高并发场景下数据库与缓存的数据不一致:

出现场景：

* 读写并发请求,导致数据库和缓存中的数据不一致,在写请求删除缓存,修改数据库库存还未成功时,查询库存的请求就发来,先去缓存中查询,发现是空,然后去数据库中查,然后将结果放入缓存中,然后修改库存的操作成功,导致数据库中是新数据,缓存中是旧数据

解决思路:

* 相同的商品id,进行哈希取值,再加上对内存队列的数量进行取模,每个商品都可以路由到某一个内存队列中,然后将去请求和写请求串行化,这样就可保证一定不会出现不一致的情况,但是会导致系统的吞吐量会大幅降低

缓存并发竞争，同一时间，多个线程来执行，操作同一个key。

解决思路:

* 方式一：分布式锁,（zookeeper分布式锁），确保同一时间,只能由一个系统实例在操作某个key,别的实例不被允许读和写
* 方式二：乐观锁 ，每次要写之前,先判断这个value的时间戳是否比缓存里的更新,如果新就允许写.

1. **为什么redis是单线程的但是还可以支撑高并发?**

* 纯内存操作【主要】
* 是基于非阻塞的IO多路复用机制,由监听程序轮询等待的事件,然后压入队列,可以达到一个线程同时处理多个io请求的目的【IO多路复用机制】
* 单线程避免了多线程的频繁切换问题

1. **Redis由哪些数据类型,分别在哪些场景下使用?**

* String,存储基本的类型
* List,有序列表,比如之前项目中的广告缓存,用的就是list集合进行缓存的,它可以用于比如存储粉丝列表,文章的评论列表等等
* Set,无序列表,它最主要的特点就是去重,比如微博上查询两个明星的共同粉丝,就可以用set进行去重
* SortSet,它主要是在set的基础上加上了排序功能
* Hash,类似map的集合,一般可以用来存储对象,我们系统中的购物车对象就是以hash类型存储在redis中,key就是用户的id,value就是购物车对象

1. **Redis过期策略**

* 定期删除配合惰性删除【主动删除+被动删除】
* 定期删除：1s中执行10【可配置】次删除任务，总占用时长250ms【可配置】，每次删除任务时长25ms： 循环执行随机抽样，删除过期的数据；单次循环何时结束：时间到了或者抽样的过期比例小于指定值【25%】。

总结：循环操作，抽样删除。绝不是全部key的扫描，代价太大。

* 惰性删除,就是当你获取key时,redis就会检查一下,如果key过期了,就删除,不会返回,如果没过期就返回

过期的时间如何存储：hash 【设置了过期时间的key：过期时间】

1. **内存淘汰策略：**

Redis内存满了怎么办，Redis使用内存淘汰机制来解决了这个问题。

* noevication:当内存不足以容纳写入新数据时,新写入操作就会报错
* allkeys-lru:当内存不足以容纳新写入数据时,在键空间中,移除最近最少使用的key
* allkeys-lfu：从所有键中驱逐使用频率最少的键
* allkeys-random:在键空间中,随机移除某个key
* volatile-lru:在设置了过期时间的键空间中,移除最近最少使用的
* volatile-lfu：从所有配置了过期时间的键中驱逐使用频率最少的键
* volatile-random:在设置了过期时间的键空间中,随机移除某个key
* volatile-ttl:在设置了过期时间的键空间中,快过期的key优先移除

1. **怎样做redis支撑高并发(读多写少用缓存,读少写多用队列)**

主从+哨兵 或者 分片集群

主从数据同步：全量同步 + 增量同步

哨兵作用：监控【主管下线+客观下线】qorum

故障转移：

1、选择话事人【哨兵，谁先发现】

2、开始故障转移：从slave节点中选择一个新的主【策略：排除响应差的、偏移量和master保持一直的】；新的主执行slaveof no one ；其他从执行slaveof 新主 ；挂点的主配置文件强制写入 slaveof 新主。

通知：java客户端，连接的是哨兵的地址。

分片集群：

数据如何分布：散列槽【16384个】

1. **redis的持久化**

如果没有持久化,遇到灾难性故障时,就会丢失所有数据，如果开启持久化到磁盘,定期备份到云服务上,就能保证遇到灾难性故障,就不会丢失全部数据

* RDB

当redis需要做持久化时,redis会fork一个子进程,将数据写到磁盘写上一个临时RDB文件中,当子进程完成写临时文件后,将原来的RDB换掉

优势:

适合做冷备份,在最坏的情况下,恢复数据要比AOF快

对redis对外提供读写服务,影响服务非常小

RDB数据快照文件更少

* AOF

可以做到更精细的持久化,redis每执行一个修改数据的命令,都会将它添加到os cache中,一般会每个一秒执行一次fsync操作,保证将os cache中的数据写入磁盘中AOF日志文件中,AOF日志文件以append-only模式写入,文件不容易破损,在AOF日志过大时,就会出现后台重写

优势:

AOF可以更好的保护数据不丢失,丢失的数据更少

AOF日志文件以append-only模式写入,没有磁盘寻址的开销,写入性能非常高

AOF日志过大时,也不会影响客户端的读写

* RDB和AOF到底如何抉择

综合使用两种持久化机制,用AOF保证数据不丢失,作为数据恢复的第一选择,用RDB来做不同程度的冷备份,在AOF文件丢失或者损坏不可用的情况下,用RDB来进行快速的数据回复

1. **redis怎么设置缓存大小【32g】**

打开redis配置文件

示例：maxmemory 100mb

单位：mb,gb。

默认为0，没有指定最大缓存，如果有新的数据添加，吃满服务器的物理内存，直到超过最大内存，则会使redis崩溃，所以一点要设置。

设置maxmemory之后，配合的要设置缓存数据回收策略, 可以通过设置LRU算法来删除部分key，释放空间(LRU是Least Recently Used 近期最少使用算法。)。

就可以修改缓存大小为16gb.

建议是物理内存的50%以下。

1. **redis集群主从数据如何同步**

全量复制 + 增量复制

1. **Redis管道【pipeline】**

批量执行redis命令，结果批量返回。

1. **Redis支持事务，但是事务不能回滚，只能放弃执行**

Multi：开启事务

Exec：执行事务

Discard：放弃执行事务

# 微服务架构

## SpringCloud常用组件有哪些？

* 注册中心：Eureka、Nacos、Consoul
* 配置中心：Nacos、SpringCloud Config
* 网关：Zuul、SpringCloud Gateway
* 服务调用：Feign、Dubbo
* 负载均衡：Ribbon
* 微服务保护：服务熔断/降级：Hystrix

## 注册中心

#### eureka和nacos的区别

* Nacos与eureka的共同点

- 都支持服务注册和服务拉取

- 都支持服务提供者心跳方式做健康检测

* Nacos与Eureka的区别

- Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式

- 临时实例心跳不正常会被剔除，非临时实例则不会被剔除

- Nacos支持服务列表变更的消息推送模式，服务列表更新更及时

- Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式

#### Eureka的工作原理

[Eureka注册中心原理 - 简书 (jianshu.com)](https://www.jianshu.com/p/caffafae2e36)

#### Zookeeper

详见zookeeper课程

## 配置中心

## 网关SpringCloud Gateway

#### 网关路由包含哪些部分？

网关路由包含：id、url、断言和过滤器

#### 你们项目中网关有哪些作用？

我们项目中网关是作为客户端访问的入口，同时也在网关端进行了权限检验和限流的操作。

#### 网关过滤器有哪些类型？

局部过滤器 GatewayFilter：作用在某一个路由上

全局过滤器 GlobalFilter：作用全部路由上

#### 说说你在开发中用到了哪些断言工厂？

Predicate来自于java8的接口。Predicate接受一个输入参数，返回一个布尔值结果。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）。可以用于接口请求参数校验、判断新老数据是否有变化需要进行更新操作。

Spring Cloud Gateway内置了许多Predict，比如Cookie 路由断、Header 路由断言、Path路由断言。

其中我们在项目中使用的比较多的断言有Path路由断言

#### 网关中如何实现跨域？

在网关的yml配置文件中，通过如下配置实现

spring:

cloud:

gateway:

globalcors:

cors-configurations:

'[/\*\*]':

max-age: 3600

allowed-origin-patterns: "\*"

allowed-headers: "\*"

allow-credentials: true

allowed-methods:

- GET

- POST

- DELETE

- PUT

- OPTION

#### 网关的负载均衡是基于什么实现的？

Ribbon

#### 说说Zuul和Gateway的区别？

1、gateway对比zuul多依赖了spring-webflux，内部实现了限流、负载均衡等，扩展性也更强，但同时也限制了仅适合于Spring Cloud套件。

zuul则可以扩展至其他微服务框架中，其内部没有实现限流、负载均衡等。

2、zuul仅支持同步，

　gateway支持异步。

3、gateway线程开销少，支持各种长连接、websocket，spring官方支持，但运维复杂，

zuul编程模型简单,开发调试运维简单，有线程数限制，延迟堵塞会耗尽线程连接资源。

## 远程调用Feign

Feign和OpengFeign的区别

* 他们底层都是内置了Ribbon，去调用注册中心的服务。eign是Netflix公司写的，是SpringCloud组件中的一个轻量级RESTful的HTTP服务客户端，是SpringCloud中的第一代负载均衡客户端。
* OpenFeign是SpringCloud自己研发的，在Feign的基础上支持了Spring MVC的注解，如@RequesMapping等等。是SpringCloud中的第二代负载均衡客户端。Feign本身不支持Spring MVC的注解，使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务
* OpenFeign的@FeignClient可以解析SpringMVC的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。