# Java基础

## List和Set区别

### List：有序可重复

1. ArrayList：线程不安全(方法上没有加synchronized关键字)，基于动态数组的数据结构，查询 效率快(通过下标查询)，插入和修改效率低， new ArrayList相当于new了一个初始容量为10的object空数组，jdk1.2新增

ArrayList扩容机制：向ArrayList添加对象时，先计算容量是否适当，原对象数组加1的长度如果大于元底层数组长度，则以适当长度(原数组的一半:5)新建一个原数组的拷贝，Arrays.copyOf()方法，并修改原数组，指向这个新建数组，原数组自动放弃

arrayList线程不安全体现：多线程下报错：

java util concurrentModificationException

|  |
| --- |
|  |

arrayList线程不安全解决方案：

a: 使用vector，new Vector()。但是尽量少用，并发低

b: 使用collections：

List list = Collections.synchronizedList(new ArrayList());

c: 写时复制:

CopyOnWriteArrayList list = new CopyOnWriteArrayList();

CopyOnWrite即写时复制容器，往一个容器添加元素的时候，不直接往当前容器Object[]数组中添加，而是先将当前容器Object[]进行copy，复制出一个新的容器Object[] newElemen,

然后往新的容器里Object[] newElements添加元素,每次添加一个元素，即扩容长度为1，添加完元素后，再将原容器的引用指向新的容器setArray(newElements)。内部使用ReentrantLock的方式加锁

这样做的好处是可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前读的容器不会添加任何元素，所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器

|  |
| --- |
|  |

1. LinkedList：线程不安全，基于链式结构实现，插入和删除效率快，查询效率慢
2. Vector：线程安全但是导致并发性下降，加synchronized锁，链式结构，基于数组结构实现，除了for循环和迭代器外，还可以使用枚举遍历，jdk1.0就有

### Set：无序不可重复

1. HashSet：无序、不可重复。线程不安全，方法没有加锁。HashSet的底层是HashMap实现，初始值16，负载因子0.75。

|  |
| --- |
|  |

hashSet中add方法的实现:值为map的key，value为new Object()

|  |
| --- |
|  |

见HashMap实现原理。HashSet要求放入的对象必须实现HashCode()方法，基于哈希算法实现，所以性能优于TreeSet

线程不安全体现：参考arrayList

线程不安全解决：a: Collections.synchronizedSet()

b: CopyOnWriteArraySet set = new CopyOnWriteArraySet();

|  |
| --- |
|  |

1. TreeSet：使用二叉树进行实现。实现了SortedSet接口，因为SortedSet有排序功能，所以TreeSet也有排序功能，使用二叉树进行排序，不允许放入空值

## Map（键值对集合的接口）

### HashMap

线程不安全，允许只有一个key为null（多了覆盖），允许多个value为null，初始容量为16，在多线程（并发）中需要通过Collections.synchronizedMap(new HashMap())或者ConcurrentHashMap来实现

HashMap实现原理：HashMap基于hash原理，我们通过put()方法和get()方法来存储和获取数据，当我们将键值对传递给put方法时，它通过调用键对象的hashCode()方法来计算hash值，我们通过hash值找到元素存储的位置，当找到元素需要存储的位置时先判断该位置上有没有元素，如果没有的话就插入，如果有的话就用新的value去替换就的value，当获取值对象时，它通过键对象的equals()方法找到正确的键值对，然后将value值返回。

HashMap扩容机制：类似ArrayList扩容机制，当向HashMap对象里面不停的添加元素时，会判断当前容器的个数，如果大于或者等于阈值时，而HashMap对象内部的数组无法加载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素，Java里面是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组

### HashTable

线程安全，效率慢，使用synchronized保证线程安全，当多线程时，同一时刻只有一个线程可以访问其HashTable同步方法，其它线程会进入阻塞状态（效率低的原因），key和value均不允许为null，初始容量为11

HashTable加锁方式（线程安全原理：在put()方法和get()方法上加锁synchronized）：继承与Dictionary，实现了Map、Cloneable和Serializable接口

### Treemap

有序

### ConcurrentHashMap（锁分段技术）

线程安全的集合容器，用于多线程和并发环境中。所有访问HashTable的线程必须竞争同一把锁，假如容器里有多把锁，每一把锁作用于容器里面的一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提升并发访问效率。ConcurrentHashMap采用锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其它段的数据也能被其它线程访问

## Jdk和jre的区别

### jdk

jdk是java development kit的缩写，是java开发工具包，是程序员使用java语言编写的java程序所需的开发工具包，是提供给程序员使用的。Jdk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：jcconsole、jvisualvm等工具软件，还包含了java程序编写所需的文档和demo

### 3.2 jre

Jre是Java Runtime Environment的缩写，是java运行时环境，包含了java虚拟机，java基础类库，是使用java语言编写的程序运行所需要的软件环境，是提供给想运行java程序的用户使用的，还有所有的java类库的class文件，都在lib下，并且都打包成了jar

如果只是运行java程序，只需要安装jre就可以了，如果要编写java程序，就得安装jdk

## ==和equals的区别

### ==

==比较的是值是否相等

如果作用于基本类型变量，则直接比较其存储的“值”是否相等

如果作用于引用类型的变量，则比较的是所指向的对象的地址

### equals

equals不能作用于基本数据类型的变量，equals继承object类，string类重写了equals方法，比较的是内容是否享同

如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址

如String、Date等类对euqals方法进行了重写的话，比较的是所指向的对象的内容

### 4.3 String str = “i”与String str2 = new String(“i”)的区别

str.equals(str2) //true因为内容相同,str==str2 //false因为地址不同

String str = “i”：是指把”i”这个值在内存中的地址赋值给str，如果再有String str3 = “i”，str和str3引用的是同一个地址值，共享同一个内存。

因为String是final类型的，所以i应该是在常量池

String str2 = new String(“i”)：是指将new String(“i”)的对象地址赋值给str2，这句话的意思是新创建了一个对象。如果再有String str4 = new String(“i”)那么相当于又创建了一个对象，然后将对象的地址赋值该str4，虽然str2和str4的值是相同的，但是他们不是同一个对象。

new String(“i”)是新建对象放到堆内存中

|  |
| --- |
| String str1 = **"i"**; String str2 = **"i"**; String str3 = **new** String(str1); String str4 = **new** String(str1); System.***out***.println(str1.equals(str2));*//true* System.***out***.println(str1.equals(str3));*//true* System.***out***.println(str3.equals(str4));*//true* System.***out***.println(str1==str2);*//true* System.***out***.println(str3==str4);*//false* System.***out***.println(str1 == str3);*//false* |

## euqals和hashCode()的区别

如果两个对象相同，即equals相同，那么它们的hashCode值一定要相同，也就是为什么重写equals方法，一定要重写hashcode方法，也就是说hashcode值要和类中的成员变量挂钩，即：对象相同🡪成员变量相同🡪hashcode值一定相同。

相反，如果两个对象的hashcode相同，则两个对象不一定相同，这里的对象相同指的是用equals方法比较

## java中final关键字的作用

final关键字可以修饰三个地方：类，属性和方法

被final关键字修饰的类不能被继承

被final关键字修饰的方法不能被覆盖(重写)

被final关键字修饰的属性不能重新赋值，否则会报错

## java中操作字符串的类及区别

string：final修饰，string类的方法都是返回new String。即string对象的任何改变都不影响原对象，对字符串的操作修改都会生成新对象

stringBuffer：线程安全（效率低），对字符串的操作方法都加了关键字synchronized来保证线程安全：new stringBuffer()

stringBuilder：不保证线程安全，效率高：new stringBuilder()

## 抽象类和接口

抽象类不一定要包含抽象方法，但是包含抽象方法的类一定要申明成抽象类

抽象类不能用final修饰，当final修饰一个类时，表明该类不能被继承

接口中定义的属性必须是public static final的，方法为public abstract，这些修饰符都可以省略

接口中定义的属性值不能在实现类中改变(public static final)

如果一个类实现了接口中的部分方法，该类必须声明为抽象类。实现全部方法则不用

## Java中的io流

按流向分为：输入流(inputStream或Reader)、输出流(outputStream或Writer)

这里的流向是以程序的运行时内存为参照

按操作的数据单元类型分为：字节流(inputStream或outputStream)、字符流(reader或writer)

字节流操作的数据单元是8位的字节(byte),字符流操作的是16位的字符

系统的输入和输出System.In和System.out都是字节流

按流的角色分为：节点流和处理流

节点流是指一个程序可以向特定的节点读写数据，直接连接数据源。

这个节点最常见的是文件，类名中包含File；还可以是数组、管道、字符串。关键字分别为ByteArray/CharArray、Piped、String

处理流并不直接连接数据源，它大多情况是对已存在的节点流进行包装，是一种典型的装饰器设计模式，使用处理流是为了更方便的执行输入输出工作。如：printStream

## Files常用方法有哪些

|  |
| --- |
| 1563708539(1) |

## Java中4中引用类型

当内存不足，JVM开始进行垃圾回收

强引用（StrongReference）：默认的支持方式。强引用没有对应的实体类，使用强引用的对象就算出现内存溢出(OOM)的异常也不会被回收，也就是说该对象永远不会被垃圾回收器回收。造成内存溢出（OOM）的主要原因

强引用使我们最常见的普通对象引用，把一个对象赋给一个引用变量，这个引用变量就是强引用

软引用（SoftReference）：如果JVM内存充足，无论发生多少次GC（垃圾回收），软引用指向的对象都不会被回收，只有在发生oom（内存溢出）才会回收，很好的避免oom，适合做缓存

Map<String,SoftReference<Bitmap>> cache = new HashMap<String,SoftReference<Bitmap>>

弱引用（WeakReference）：与软引用唯一的区别：不管内存充不充足，只要发生了GC（垃圾回收），弱引用指向的对象就会被回收

虚引用（PhantomReference）：虚引用不会决定对象的生命周期，任何时候都会被回收，无法通过虚引用获取对象，get()方法总是返回null。必须和引用队列一块使用，可以将其放到引用队列里面，执行一些后续操作

使用方法：

|  |
| --- |
|  |

## http和https的区别：

http是超文本传输协议，制定了万维网和服务器之间的通信规则

https是以安全为目标的http通道，SSL加密

## cookie和session区别

session存在于服务器，cookie存在客户端，所以从安全上来讲，存在session里的数据比存在cookie里的数据安全，然后我们获取session信息是通过存放在会话cookie里的sessionId获取，因为session存放在服务器，所以session里面的东西不断增加会增加服务器的压力，所以会把一些重要的东西存放在session服务器，而把一些次要的东西放在客户端cookie中

cookie分为会话cookie和持久化cookie

会话cookie存放在浏览器的内存中，所以生命周期和浏览器是一致的，所以浏览器关了，cookie也就消失了，虽然浏览器关闭cookie会消失，sessionId也就不存在了，但是session信息还是存在服务器端，我们只是查不到session信息，但是它并不会消失。而持久化cookie存放在客户端的硬盘中，所以持久化cookie的生命周期就是我们设置的那个保存时间

session消息丢失的情况：调用session.invalidate()方法或者session.removeAttribute()方法

## Static（静态）变量

1. static不允许修饰局部变量
2. 很多时候我们将只需要进行一次初始化的操作都放在static修饰的代码块中
3. Static变量也称静态变量，静态变量和非静态变量的区别：静态变量被所有对象共享，在内存中只有一个副本，它只有在类被加载时才会被初始化。非静态方法是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响
4. 非静态方法可以调用static（静态）方法，静态方法不能调用非静态方法

## GC（垃圾回收）机制

### 15.1 概述

程序计数器、虚拟机栈、本地方法栈随线程终止而终止（因为线程私有），栈中的栈帧随着方法的进入和退出执行着入栈和出栈的操作。而java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序运行期才知道哪些对象被创建，这部分内存的分配和回收都是动态的，垃圾回收所关注的就是这部分内存。

在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下不会执行，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫描那些没有被任何引用的对象，并将它们添加到要回收的的集合中，进行回收。

好处：有效的防止内存泄露、保证内存的有效使用。

释放没有被引用的对象所占领的内存，目的在于清除不再使用的对象

### 15.2 对象是否可回收判断

（1）引用计数法：给对象添加一个引用计数器，但是难以解决循环引用的问题

（2）可达性分析法：通过一系列的‘GC Roots’对象作为起始点，从这个被称为GC Roots的对象开始向下搜索，从这些节点出发所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连的时候说明对象不可用。即给定一个集合的引用作为根出发，通过引用关系遍历对象图，能被遍历到的(可到达的)对象就被判定为存活；没有遍历到的就自然判定为死亡。

|  |
| --- |
|  |

可作为GC Roots的对象：

虚拟机栈（栈帧中的局部变量表）中引用的对象

方法区中类静态属性引用的对象

方法去中常量引用的对象

本地方法栈中JNI（一般说的Native方法）引用的对象

### 15.3 垃圾回收算法

分代回收算法：根据存活对象划分几块内存区（新生代、年轻代、永久代），其中新生代又分为一块较大的Eden区和两块大小相等的survivor区。

新生代收集器采用复制算法：将可用内存大小分为大小相等的两块，每次只是用一块，当这一块内存用完了，就将存活的对象复制到另一块，最后将此块内存一次性清理掉。可以解决内存碎片化问题，但是会造成空间利用率低下。因为大多数新生代对象都不会熬过第一次GC。所以没必要1:1规划空间。可以分为一块较大的Eden空间和两块较小的survivor空间，每次使用Eden空间和其中一块survivor空间。当回收时，将Eden和survivor中存活的对象复制到另一块survivor上，最后清理Eden和survivor空间（两块survivor空间依次交替使用）。大小比例一般是8:1:1，每次浪费10%的survivor空间，但是如果存活的对象大于10%的话，这里采用一种分配担保策略：多出来的对象直接进入老年代

老年代对象存活率较高，没有额外的空间分配对他进行担保。所以老年代收集器必须使用标记-清理和标记整理（压缩）算法：

（1）标记-清理的算法：先标记处所有需要回收的对象，标记完成后统一回收所有被标记的对象。效率低，会产生大量不连续的内存碎片，不推荐使用

（2）标记整理算法：解决标记清除内存碎片化问题，推荐使用

### 15.4 垃圾回收器

垃圾回收算法是内存回收的理论，垃圾回收器是内存回收的实践

15.4.1 如何查看服务器默认的垃圾收集器是哪个？

java -XX:+PrintCommandLineFlags -version

|  |
| --- |
|  |

15.4.2 如何选择垃圾回收器？

|  |
| --- |
|  |

15.4.3 垃圾收集器的种类：

1. Serial收集器(串行-XX:+UseSerialGC，不推荐使用)：

它为单线程环境设计且只是用一个线程进行垃圾回收，并且在垃圾回收时必须暂停其它所有的工作线程直到收集结束，新生代垃圾回收器。所以不适合服务器环境

1. Parallel Scavenge收集器(并行：java8默认UseParallelGC)

多个垃圾收集线程并行工作，此时用户线程是暂停的

这是一个新生代收集器，使用复制算法实现

目的是达到可控制的吞吐量：

吞吐量=代码运行时间/(代码运行时间+垃圾回收时间)

作为一个吞吐量优先的收集器，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整停顿时间。这就是GC的自适应调整策略

1. CMS收集器(UseConcMarkSweepGC并发垃圾回收器)：

用户线程和垃圾收集线程同时执行(不一定是并行，可能交替执行)，不需要停顿用户线程。目的是尽可能缩短垃圾回收时用户线程所停顿的时间。基于标记-清除算法实现

1. G1收集器(UseG1GC：java9默认)

G1回收器将堆内存分割成不同的区域然后并发的对其进行垃圾回收。面向服务端的垃圾回收器

优点：并行与并发、分代收集、空间整合、可预测停顿

（6）ParNew(-XX:+UseParNewGC并行，不推荐使用)

（6）ZGC(java11默认)

## Osi与TCP/IP协议簇

OSI（7层模型）

1. 物理层：为设备之间的信息传输提供可靠环境
2. 数据链路层：提供数据的传送服务，数据传输的单元是帧，不同的协议帧的长短也不一样
3. 网络层：通过路由算法选择合适的通道传输数据
4. 传输层：数据传输
5. 会话层：使应用建立和保持会话
6. 表示层：数据的表现形式，文字、图片等都有各自的格式
7. 应用层：实现应用进程间的数据交换

TCP/IP（4层模型）

1. 数据链路层：负责接收IP数据报并通过网络发送至IP层，或者从网络上接收物理帧，抽出IP数据报，交给IP层
2. 网络层：负责相邻计算机之间的通信。
3. 传输层：提供应用程序间的通信
4. 应用层：向用户提供一组常用的应用程序，比如：电子邮件

TCP/IP的建立（3次握手）

SYN：请求建立连接

ACK：确认号是否有效，一般为1

FIN：希望断开连接

第一次握手：客户端向服务器发送SYN报文(SYN一般是用来同步的)，此时客户端进入SYN\_SEND状态，服务器进入SYN\_RECE等待确认状态

第二次握手：服务端接收到SYN报文后同时向客户端发送接收确认报文，并带上ACK报文(用于响应)

第三次握手：客户端接收到ACK报文后并检查ACK报文是否正确，如果正确的话客户端再次发送ACK，服务器收到后确认验证通过，表示连接已经成功建立，可以发送数据包了

TCP/IP建立的释放（4次挥手）

第一次挥手：客户端向服务器发送FIN，用来告诉服务器你发的东西我收到了，客户端要关闭通道

第二次挥手：服务器从客户端收到FIN，并发回一个ACK报文(用于应答)，确认收到客户端的消息

第三次挥手：服务器关闭与客户端的连接，发送FIN给客户端

第四次挥手：客户端收到后发回ACK报文确认

## Java反射

反射的定义及作用：Java反射就是在运行状态中，对于任意一个类，我们都能获取到里面对应的属性和方法；对于任意一个对象，我们都能够调用它的任意属性和方法，并且改变它的属性；通过继承，我们也可以获取父类的属性

使用方式

得到class的3种方式

1. Class class1 = 对象.getClass();
2. Class class2 = 类.class();
3. Class class3 = Class.forName(“com.liu.bean.Person”);

然后通过class对象获取成员变量、接口、构造方法、普通方法

getName()：获取类的完整名字

getFields()/getDeclareFields()：获取类的public/private类型的属性

getMethod(String 方法名,Class[] 方法的参数类型)：获取方法

总结：需要合理使用，反射会消耗系统的性能，增加复杂性

应用：Mybatis的结果集是通过反射来实现的

## Arrays.sort和Collections.sort的实现原理

Collections.sort()实际会将List转为数组，然后调用Arrays.sort()，排序完了再转回List。而Arrays.sort()的底层是TimeSort实现的

## Collection和Collections

Collection是接口，list和set均实现此接口

Collections是一个帮助类，可以实现list和set的线程安全

## Cloneable接口实现原理

Cloneable是一个标记接口，该接口里面没有方法，只需实现就可。

但是想要使用对象的clone方法，必须实现此接口implemetns，23种设计模式之一原型模式就实现此接口

obj.clone != obj 为true，他们不是一个对象，地址不同

obj.clone.getClass()==x.getClass() 为true，它们是同一个类型的Class

obj.clone.equals(obj) 为true

## linkedHashMap的应用

LinkedHashMap（HashMap+LinedList）是有序的，而HashMap是无序的，所以有时候我们需要使用一个有序的map的时候，就可以使用LinkedHashMap，都是非线程安全的

## 23种设计模式

个人理解：都是为了简化代码，重用代码，减少代码的复杂性

创建型模式（5种）：单例模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式

结构型模式（7种）：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式

行为型模式（11种）：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

设计模式6大原则：

1. 开闭原则：为了程序扩展性，易于升级和维护
2. 里氏代换原则
3. 依赖倒转原则
4. 接口隔离原则
5. 迪米特法则
6. 合成复用原则

## Servlet的运行原理及生命周期

Servlet是一个应用程序，运行在服务器端，用来处理客户端请求并作出响应的程序

Servlet是多线程的，它的生命周期是由web容器（Tomcat）负责的

Servlet运行原理：

1. 客户端向服务器端发送请求
2. 这时tomcat会创建两个对象，httpServletRequest请求和httpServletResponse响应
3. Servlet根据请求路径找到web.xml中的servlet的配置信息，然后创建servlet对象
4. 执行请求，调用servlet的service()方法，并将事先创建好的request对象和response对象作为service()方法的参数传递给servlet对象
5. Servlet可以通过request获得请求参数，进行相应的处理，然后将返回结果写到response对象上
6. 容器读取response对象上的数据，然后将处理结果打包（响应数据包）发送给客户端

Servlet生命周期：

实例化：创建servlet对象，在web.xml配置文件中配置load-on-startup的值大于0，tomcat容器启动时实例化；否则，调用时实例化

初始化：调用servlet的init()方法

处理请求：调用servlet的service()方法

销毁：调用servlet的destroy()方法

## 类加载过程及生命周期

类加载5个过程：

加载：类加载器获二进制字节流，将静态存储结构转化为方法区的运行时数据结构，并生成此类的Class对象

验证：验证文件格式、元数据、字节码、符号引用，确保Class的字节流中包含的信息符合当前虚拟机的要求

准备：为类变量分配内存并设置初始化值，这些变量使用的内存都将在方法区中进行分配

解析：将常量池内的符号引用替换为直接引用，包括类或接口的解析、字段解析、类方法解析、接口方法解析

初始化:执行类中定义的字节码文件（Java代码）

类的生命周期：

加载、验证、准备、解析、初始化、使用、卸载

## Java类加载器及如何加载类

类加载器是实现通过一个类的全限定名来获取描述此类的二进制文件流的代码模块。类加载器是通过双亲委派模型来完成的

双亲委派模型工作过程：如果一个类加载器接收到类加载的请求，它会先把这个请求委派给父加载器去完成，只有父加载器反馈自己无法完成加载请求时，子加载器才会尝试自己去加载。所有的加载请求最终都会传送到启动类加载器中。

使用双亲委派模型的好处：能够有效确保一个类的全局唯一性，当程序中出现多个限定名相同的类时，类加载器在执行加载时，始终只会加载其中的某一个类

## 冒泡排序

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] arr={6,3,8,2,9,1};  **for**(**int** i=0;i<arr.length-1;i++){//外层循环控制排序趟数  **for**(**int** j=0;j<arr.length-i-1;j++){//内层循环控制每一趟排序次数  **if**(arr[j]<arr[j+1]){ //<：倒序 >：正序  **int** temp = arr[j];  arr[j]=arr[j+1];  arr[j+1]=temp;  }  }  }  //排序后的数组  **for** (**int** i : arr) {  System.*out*.println(i);//986321  }  } |

# Java并发

## synchronized的实现方式以及锁优化

### 1.1 为什么要用synchronize以及实现方式

使用synchronizde关键字的原因：线程安全是并发过程中的重点：造成线程安全的原因有两点，当存在多个线程操作共享数据时，需要保证同一时刻有且只有一个线程在操作共享数据，其它线程必须等该线程处理完再进行，这种方式叫做互斥锁，也就是说当一个共享数据被正在访问的线程加上互斥锁后，同一个时刻，其他线程只能出于等待状态，只当该线程释放锁，synchronized可以保证同一个时刻只有一个线程访问某个方法或者某个代码块

三种使用方法：当一个线程访问同步代码块时，它首先需要得到锁才能执行同步代码块，当退出或者抛异常时释放锁。

（1）修饰普通方法，当前线程的锁是实例对象increase：

|  |
| --- |
|  |

（2）修饰静态方法，锁是当前类的class对象：

|  |
| --- |
|  |

（3）修饰同步代码块，锁是当前括号里面的对象，可用this代替：

|  |
| --- |
|  |

我们也可以使用this（当前实例）或者当前类的class对象作为锁

|  |
| --- |
|  |

### 1.2 锁优化

4种锁状态：无锁状态、偏向锁、轻量级锁和重量级锁

偏向锁（1.6之后）核心思想：如果一个线程获得锁，就进入偏向模式，Mark Word结构也变为偏向锁结构，当这个线程再次获取锁时，无需再做重复获取锁的操作，这样就省去了大量操作，提升性能。

轻量级锁（1.6之后）：当偏向锁失败后，虚拟机并不会立即变成重量级锁，而是尝试使用轻量级锁的优化手段，此时Mark Word也变为轻量级的锁结构

自旋锁：轻量级锁失败后，进行称为自旋锁的优化手段，空循环（50到100个之间），如果在此期间得到锁，顺利进入临界区，如果还不能获得锁，升为重量级锁

锁消除：去除不可能存在竞争共享资源的锁，节省没必要的请求锁时间

|  |
| --- |
|  |

### 1.3 synchronized的可重入性

一个线程得到一个对象锁后再次请求该对象锁，是允许的，也就是：当前方法和方法里面调用的方法可以同时加synchronized

## Java锁

### 公平锁和非公平锁

* + 1. 公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁。先来后到

公平锁Lock lock = new ReentrantLock(true);

* + 1. 非公平锁

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。在高并发的情况下，有可能造成优先级反转或者饥饿(某个线程一致获取不到锁)现象

非公平锁Lock lock = new ReentrantLock(false);

ReentrantLock默认非公平锁Lock lock = new ReentrantLock();

Synchronized锁也是非公平锁

### 可重入锁（递归锁）

指的是同一线程外层函数获得锁之后，内层递归函数仍然能获取该锁的代码。一个线程获取一个方法的锁，则获取这个方法里面所有(方法或者代码块等)的锁。也就是说：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

ReentrantLock和synchronized是可重入锁(非公平的可重入锁)

作用：避免死锁（死锁参考第9点）

### 自旋锁

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗cpu

实现：原子引用+CAS

### 读(共享锁)写(独占锁)锁/互斥锁

独占锁：指该锁一次只能被一个线程所持有。

ReentrantLock和Synchronized都是独占锁

共享锁：该锁可以被多个线程所持有

ReentrantLockReadWriteLock中读锁是共享锁，写锁是独占锁

读锁的共享锁可保证并发读是非常高效的，写写、读写、写读的过程是互斥的

## Volatile的实现原理

Volatile是java虚拟机提供的轻量级的同步机制

### volatile的作用

多线程访问共享变量，为了保证共享变量能够被一致和准确性的读取和更新，线程应该通过互斥锁（排他锁）单独获得这个变量，用volatile修饰变量或者字段，可以保证所有线程看到的这个变量是一致的，比锁更加方便

### volatile的三大特性(使用)

1. 有序性（防重排序、禁止指令重排）

单线程环境里面确保程序最终执行结果和代码顺序执行的结果一致。

处理器在进行指令重排序时必须考虑指令间的数据依赖性

多线程环境中线程交替执行，由于编译器优化重排、指令并行的重排、内存系统的重排的存在，两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。因为步骤2和步骤3不存在数据依赖性，所以可能会发生指令重排

Public static volatile Dog dog；

实例化对象的步骤：1）分配内存空间

2）初始化对象

3）将内存空间的地址赋值给对应的应用

但是由于操作指令对操作进行重排序，上面的过程会被打乱，多线程可能会将一个未初始化的对象引用暴露出来，造成异常，为了防止上述重排序，我们将变量设置为volatile类型的变量

1. 实现可见性

可见性指：当存在多个线程的时候，其中一个线程修改了共享变量值，由于每一个线程都有自己的线程工作内存，而另一个线程看不到修改过或者看到的是未修改的值，当一个线程修改了主内存中的值后，要通知其它线程值已经失效，需要去主内存中重新获取。volatile可以有效的解决这个问题

1. 不保证原子性

volatile对变量的单次读写可以保证其原子性，但是i++不能保证，因为它是读、写两次操作。

当多线程的时候，一个线程2刚把自己工作内存中的值刷新到主内存时，还没来得及通知其他线程，其他线程1也立马把自己的值刷新到主内存，这时会造成线程1的值将线程2的值覆盖，造成线程2的值丢失。

解决不保证原子性的方法：

A．加synchronized关键字保证原子性

B．使用原子类atomic，atomicInteger

原子性、可见性、有序性也是多线程三大特性

### volatile实现原理

实现可见性：（1）修改volatile变量时会将强制修改后的值刷 新到主内存（内存条）中

1. 修改volatile变量后会导致其他线程工作内存中对应的变量值失效，因此在读取该变量值的时候就需要重新从主内存中读取

由于JVM运行实体的程序是线程，而每个线程创建时JVM都会为其创建一个自己的工作内存（或者叫做栈空间），工作内存是每个线程私有的数据区域，而JAVA内存模型中规定所有变量都存储在主内存。主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作（读取赋值等）必须在自己的工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作。操作完成后再将变量写回主内存。不能直接操作主内存中的变量，各个线程的工作内存中存储着主内存中的变量副本拷贝，因此不同的线程无法访问对方的工作内存，线程间的通信（传值）必须通过主内存来完成

### 2.4 你在哪些地方用到过volatile

（1）单例模式：参考第21条

## Java信号灯Semaphore

作用：1 用于多个共享资源的互斥使用2 用于并发线程数的控制

当我们创建一个可扩展大小的线程池，并且需要在线程池内同时让有限的线程并发运行时，这就需要用到semaphore（信号灯机制），semaphore通常用于限制访问某些资源（物理或逻辑）的线程数目，他是一个信号计数量，从概念上讲，信号量维护了一个许可集合，在线程执行前会阻塞每一个线程，然后再从信号量获取许可，线程开始执行，在线程执行完后，每个线程释放许可，从而可能释放一个正在阻塞的获取者。

在线程池内创建线程运行时，每个线程必须从信号量获取许可，从而保证可以使用该项，该线程结束后，线程返回到池中并将许可返回到该信号量，从而允许其他线程获取该项。注意：调用acquire()时无法保证同步锁定，因为这会阻止线程返回到池中，信号量封装所需的同步，以限制对池的反问，这同维持该池本身一致性所需的同步是分开的

|  |
| --- |
|  |

## 如何实现所有线程在等待某个事件发生才会执行

1. 信号量semaphore：设置信号量的初始值为等待的线程数N,一开始将信号量申请完，让剩余的信号量为0，待事件发生时，同时释放N个占用的信号量，则等待信号量的所有线程将获取信号量得以继续执行
2. 闭锁CountDownLatch：闭锁是典型的等待事件发生的同步工具类，将闭锁的初始值设置为1，所有线程调用await方法等待，当事件发生时调用CountDown将闭锁值减为0，则所有await等待闭锁的线程得以继续执行。

某一事件所有线程都执行完在执行：

|  |
| --- |
|  |

1. 阻塞队列BlockingQueue：所有等待事件的线程尝试从空的阻塞队列获取元素，将阻塞，当事件发生时，向阻塞队列中同时放入N个元素（N的值与等待的线程数相同），则所有等待的线程从阻塞队列中取出元素后得以继续执行
2. 栏CyclicBarrier：设置栅栏的初始值为1，当事件发生时，调用barrier.wait()冲破设置的栅栏，将调用指定的Runnable线程执行，在该线程中启动N个新的子线程执行。注意：这个方法并不是让执行中的线程全部等待在某个点，待某一事件发生后继续执行，而是相当于启用新的线程执行

某一事件所有线程都执行完在执行：

|  |
| --- |
|  |

## 什么是CAS?CAS有什么缺陷，如何解决

CAS(Compare And Swap)：比较交换，它是一条CPU并发原语

它的功能是判断内存某个位置的值是否为预期值，如果是，则更改为新的值，这个过程是原子的

乐观锁：认为事情总是会往好的方向发展，总是认为坏的情况发生的概率特别小，可以无所顾忌的做事。

悲观锁：总是认为发展事态如果不及时控制，以后就无法挽回，即使无法挽回的局面不可能发生。

悲观锁的实现：sql加for update

乐观锁和悲观锁就如同并发过程中的加锁与无锁的策略，即加锁是一种悲观策略，无锁是一种乐观策略，因为对于加锁的并发程序来说，他们总是认为每次访问共享资源时总会发生冲突，因此必须对每一次数据操作实施加锁策略。而无锁总是假设访问共享资源不会发生冲突，线程可以不停执行，无需加锁，无需等待，一旦发生冲突，无锁策略采取一种称为CAS的技术来保证线程执行的安全性，CAS技术是无锁策略实现的关键，因此不会出现死锁：天生免疫死锁。

原理：

CAS(compareAndSet)：比较交换，底层使用unsafe来实现

|  |
| --- |
|  |

|  |
| --- |
|  |

执行函数（V,E,N）：源码参考第18条

V：表示要更新的变量E：表示预期值N：表示新值

如果V跟E的值一样，则将V值设为N，返回true，如果V跟E的值不一样，则说明已经有其它线程做了更新，返回false，当前线程什么都不做。即CAS需要我们提供一个预期值，当预期值与当前线程变量值相同时，说明还没线程修改该值，当前线程可以进行修改，也就是执行CAS操作，但如果预期值与当前线程不符，则说明该值已经被其他线程修改，此时不执行更新操作，但是他会循环重新读取该变量再尝试再次修改该变量，直至修改成功。源码：do…while循环

|  |
| --- |
|  |

当多个线程同时使用CAS操作一个变量时，只有一个更新成功，其余线程均失败，但失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试。

CAS是一些CPU直接支持的指令（CPU并发原语），并发原语的执行必须是连续的，在执行过程中不允许被中断，不会造成数据不一致问题，所以保证了原子性。在JAVA中无锁操作基于以下3个方法实现（重写）：

|  |
| --- |
|  |

问题及解决方法：

1. 如果CAS失败，会一直进行尝试。如果CAS长时间一致不成功，一致循环会给CPU带来很大的开销
2. 只能保证一个共享变量的原子操作
3. ABA问题：加入有两个线程，从主内存中获取的值是5到自己的工作内存，假如线程1将自己的工作内存中的值从5变成50，并写回主内存，后又将主内存的50改为5，那么当线程2去更新值的时候发现预期值还是5，便将自己修改的值写回主内存。虽然预期值还是5，但是它是被线程1修改两次后变为的5，这就是ABA问题。（当V与E的值相同时，V被更新为N值的过程中，另外两个线程连续修改了两次变量V的值，这样的话又使该值恢复到旧值）问题。可以通过原子引用（**AtomicStampedReference：带时间戳的原子引用即**新增修改时间，判断自己的获取时间与最新修改时间是否一致）来解决，参考18.2

## Synchronized和Lock的区别与使用

|  |  |  |
| --- | --- | --- |
|  | Synchronized | Lock |
| 存在层次及实现原理 | 属于jvm层面，是Java关键字，在object下面。底层通过moniter实现(一次进入：moniterEnter，两次退出：正常退出和异常退出：moniterExit)，修饰普通方法、静态方法以及代码块 | 属于API层面，jdk1.5之后的一个类  Lock lock = New ReentrantLock(); |
| 锁的获取 | 假设A线程获取锁，B线程等待，如果A线程阻塞，B线程会一直等待 | Lock有多个获取锁方式，可以尝试获得锁，线程不用一直等待。lock和unLock方法配合try/finally方法语句块完成且必须成对出现，即一个方法里可以有多个lock和多个unLock，数量上要一样 |
| 锁的释放 | 1.获取锁的线程执行完同步代码释放锁2.线程执行异常，Jvm会让线程释放锁 | 在finally中必须释放锁（lock.unLock()），要不容易造成线程死锁 |
| 锁状态 | 无法判断 | 可以判断lock.tryLock()，如果返回false，锁占用，返回true，表示可以获取锁 |
| 等待是否可中断 | Synchronized不可中断，除非抛出一场场或者正常完成 | 可以中断，设置超时方法  1：tryLock(Long time,TimeUnit unit)  2：lockInterruptibly()放代码块中，调用interruput()方法可中断 |
| 加锁是否公平 | 非公平锁 | 默认非公平锁，可以使用公平锁  new ReentrantLock(true/false) |
| 锁绑定多个条件Condition | 没有，要么随机唤醒，要么全部唤醒 | Reentrant用来实现分组唤醒需要唤醒的线程，可以精确唤醒 |

## AQS

AQS（AbstractQueuedSynchronizer）：队列同步器，它是Java并发用来构建锁和其它同步组件的基础框架。它是一个抽象类，主要通过继承来使用：

|  |
| --- |
|  |

原理：AQS的实现依赖内部的同步队列（FIFO双向队列），如果当前线程获取同步状态失败，AQS会将该线程以及等待状态等信息构成一个节点（Node），将其加入同步队列的尾部，同时阻塞当前线程，当同步状态释放时，唤醒队列的头节点。

AQS最主要的三个成员变量：

|  |
| --- |
|  |

获取同步状态：假设A线程获取同步状态（获取锁），初始状态state=0，A线程获取锁，将state置为1，表示锁被占用，所以将B线程信息和等待状态等信息构成一个Node节点对象，放入同步队列，同时阻塞线程B（）。

释放同步状态：线程A释放锁时，将state置为0，此时A会唤醒头节点的后续节点，此时B可以获取锁

## 什么是死锁以及死锁的检测与预防

定义：

死锁是指两个或两个以上的进程在执行过程中因争夺资源而造成的一种互相等待的现象，若无外力作用，他们将无法推进下去，此时称系统发生了死锁或者处于死锁状态。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁

产生原因：

1. 系统资源分配不足
2. 进程运行推进的顺序不合适
3. 资源分配不当

产生死锁的4个必要条件：

1. 互斥条件：同一时间不止有一个线程操作共享数据
2. 请求与保持条件：一个线程因请求资源发生阻塞时，对已获取的资源保持不放
3. 不剥夺条件：线程获取到资源且在进程释放之前，只能自己用完释放，不能被剥夺
4. 循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路，造成永久阻塞

如何避免死锁：

1. 加锁时限：若一个线程没有在给定的时间内成功获得所需要的锁，则会进行回退并释放所有已获得的锁
2. 死锁检测：

死锁定位及解决：

1. jps命令定位进程号：jps -l
2. jstack找到死锁查看：jstack 2112(进程号)

|  |
| --- |
|  |

## 多线程

并发（Concurrent）：用户线程和GC线程同时执行，不需要停顿用户线程，垃圾回收线程在另一个CPU上运行

并行（Parallel）：多个GC线程（垃圾收集线程）并行工作，此时用户线程暂停执行（时间短）

进程：进程是操作系统资源分配的基本单位，一个进程包含多个线程，即多线程

线程：线程是任务调度和执行的基本单位

### 10.1 线程的状态

新建状态：new之后，start或者run之前

就绪状态：线程对象创建后，调用该对象的start()方法或者run()方法，该线程位于可运行的线程池中，变成可运行状态，等待获取CPU的使用权

运行状态：就绪状态的线程获取了CPU的使用权，开始执行代码

阻塞状态：由于某种原因放弃了CPU的使用权，进入阻塞状态

死亡状态：退出了run()方法

### 10.2 多线程的实现方式

1. 实现Runnable接口，线程执行完没有返回值
2. 继承Thread类，Thread类也是Runnable接口的子类，Thread类实现Runnable接口，线程执行完没有返回值
3. 实现callable接口，FutureTask实现了Runnable接口，而new FutureTask()需要Callable的实现作为构造参数，所以通过FutureTask包装器来创建Thread线程，线程执行完有返回值，需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取结果，如果需要获取callable线程的计算结果，将其放到最后，不影响主线程的执行；当不掉用此方法时，主线程不会阻塞。如果线程出现异常，FurureTask.get()方法会抛出异常(throws interruptedException或者ExectionExection)。如果线程取消，会抛出CancellationExection
4. 线程池

### 10.3 Runnable和Callable的区别

1. Runnable执行方法是run(),Callable执行方法是call()
2. 实现Runnable接口的任务线程无返回值，实现Callable接口的任务线程能返回执行结果
3. Call方法可以抛异常，run方法有异常只能在内部消化

## 线程池的用法及好处

### 11.1 什么是线程池

线程池是指在初始化一个多线程应用程序中创建一个线程集合，然后在需要执行新的任务时重用这些线程而不是新建线程

线程池做的工作主要是控制线程的运行数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量则超出数量的线程排队等候，等其他线程执行完毕，再从队列中取出任务执行

它的主要特点是：线程复用、控制最大并发数、管理线程

### 11.2 ThreadPool的用法

Java中的线程池是通过Executor框架实现的，该框架中用到了Executor、Executors、ExecutorService、ThreadPoolExecutor这几个类。

ExecutorService继承了Executor接口，AbstractExecutorService实现了ExecutorService接口，ThreadPoolExecutor继承了AbstractExecutorService类。

new ThreadPoolExecutor()提供了4个构造方法，7个参数的含义：

|  |
| --- |
|  |

corePoolSize(int)：线程池中核心线程数，一般为CPU数量

maximumPoolSize(int)：线程池中线程总数，一般为CPU数量()\*2，线程池中最多能创建的线程数量

keepAliveTime(long)：线程池中非核心线程闲置超时时长，默认情况下只有线程数大于corePoolSize才会起作用，可以设置

Uint(TimeUint)：keepAliveTime的单位

workQueue(BlockingQueue)：线程池中的任务队列，维护着等待执行的Runnable对象，即被提交但尚未被执行的任务

threadFactory(ThreadFactory)：生成线程池中工作线程的线程工厂，主要用来创建线程，一般用默认的即可Executors.defaultThreadFactory

Handler(RejectedExecutorHandler)：拒绝任务处理时的策略

拒绝任务处理的4种策略均实现了RejectedExecutorHandler接口

ThreadPoolExecutor.AbortPolicy：默认策略，新任务提交时直接抛出未检查异常(RejectedExecutionException)阻止系统正常运行。当任务数达到maximumPoolSize+队列的限制后，会报异常

ThreadPoolExecutor.DiscardPolicy：直接丢弃任务，不予任何处理也不抛出异常。如果允许任务丢失，这是最好的一种方案

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面(等待最久)的任务，然后把当前任务加入队列中尝试再次提交任务

ThreadPoolExecutor.CallerRunsPolicy：为调节机制，既不抛弃任务也不抛出异常，而是将某些任务回退到调用者，由调用线程处理该任务，从而降低新任务的流量。任务会执行，只不过由调用者执行

### 11.3 使用线程池的好处

1. 降低资源消耗：通过重复利用已创建的线程，降低线程创建和销毁造成的资源消耗
2. 提高响应速度：当任务到达时，任务可以不需要等到线程的创建就能执行
3. 提高线程的管理性：线程时稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配、调优和监控
4. 线程池允许我们开启多个任务而不用为每个线程设置属性线程池状态

## ThreadLocal

概念：ThreadLocal为解决多线程程序的并发问题提供了一种新的思路，但是ThreadLocal无法解决共享对象的更新问题（也就保证了线程的安全性）

使用场景：ThreadLocal是用在多线程的场景的

1. 线程安全的（由于不需要共享信息，也就不存在竞争问题，从而保证了线程的安全），保存线程上下文信息，在任意调用的地方可以获取，多个线程需要使用同一个变量，但这个变量的值不需要再各个线程间共享，各个线程都只使用自己的这个变量的值
2. 使用ThreadLocal来减少参数的传递，用ThreadLocal定义全局变量，各个线程需要参数时，去全局变量去取就可以了

实现原理：Thread类有属性变量threadLocals（类型是ThreadLocal.ThreadLocalMap），也就是说每个线程有一个自己的ThreadLocalMap，所以每个线程往这个ThreadLocalMap中读写隔离的，并且是互不影响的。ThreadLocalMap中Key是ThreadLocal的对象，value是对应的值

常用方法：set(),get(),remove()

## 线程池中submit和execute方法的区别

1. 接受的参数不一样
2. Submit提交指定的任务去执行并且返回Future对象，有返回值，如果需要知道任务的执行结果可以用submit。execute开启线程执行池中的任务，没有返回值
3. Submit可以抛出异常，方便用户捕捉处理，Future.get()

## 线程池的状态

Running、ShutDown、Stop、Tidying、Terminated

### Running

1. 状态说明：线程池处于Running状态时，能够接收新任务，以及对添加的任务进行处理
2. 状态切换：线程池的初始化状态就是Running。换句话说，线程池一旦被创建，就处于Running状态，并且线程池中的任务数为0

### ShutDown

1. 状态说明：线程池处于ShutDown状态时，不接收新任务，但能处理已添加的任务
2. 状态切换：调用线程池的ShutDown()接口时，线程池由Running -> ShutDown

### Stop

1. 状态说明：线程池处于Stop状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务
2. 状态切换：调用线程池的ShutDownNow()接口时，线程池由(Running/ShutDown) -> Stop

### Tidying

1. 状态说明：当所有的任务已终止，ctl记录的任务数量为0，线程池会变为Tidying状态，当线程池变为Tidying状态时，会执行钩子函数terminated()。Termnated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为Tidying时，进行相应的处理；可以通过重载terminated()函数来实现
2. 状态切换：当线程池在ShutDown状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由ShutDown->Tidying

当线程池在Stop状态下，线程池中执行的任务为空时，就会由Stop -> Tidying

### Terminated

1. 状态说明：线程池彻底终止，就变成Terminated状态
2. 状态切换：线程池处在Tidying状态时，执行完terminated()之后，就会由Tidying -> Terminated

## 线程池的种类、区别、应用场景、创建方式

（1）newCachedThreadPool

区别：创建一个可根据需要创建新线程的线程池，但是以前构造的线程在可用时也将重用他们。当有新任务进来的时候，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此调用execute时会在池中寻找可用线程来执行，若有可用线程则执行，若没有可用线程将创建一个新线程来执行该任务，若池中空闲线程空闲时间超过指定大小，则该线程会被销毁。

使用场景：执行很多短期异步任务的程序或者负载较轻的服务器使用该线程可提高程序性能

|  |
| --- |
|  |

（2）newFixedThreadPool

区别：创建一个指定线程数量的线程池，每当提交一个任务就创建一个新的线程，如果线程数量达到线程池初始的最大数，则将提交的任务添加到池队列（阻塞队列）中

应用场景：执行长期的任务可以提高程序性能

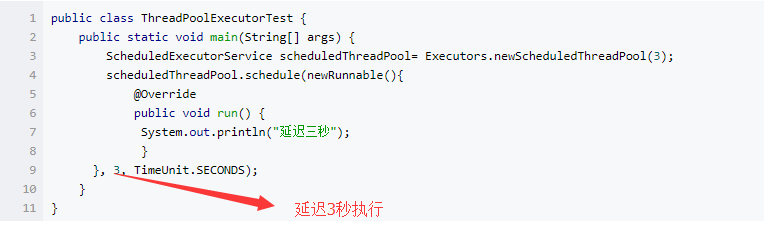
|  |
| --- |
|  |

（3）newScheduledThreadPool

区别：创建一个可以定时的线程池，可以设置线程指定时间后执行命令或者定期执行，如果线程池内线程均处于繁忙状态，则新任务会进入DelayedWorkQueue队列中

应用场景：周期性执行任务、延迟指定时间后执行任务或者定期执行任务的场景

延迟执行代码示例：



延迟指定时间后定期执行代码示例：



（4）newSingleThreadExecutor

区别：创建一个只用一个线程的线程池

应用场景：一个任务一个任务执行代码的场景

|  |
| --- |
|  |

（5）newWorkStealingPool(int)

Java8新增，使用目前机器上可用的处理器作为他的并行级别

注意：生产中不能直接使用Executors.new创建线程池，因为队列如果使用linked等，请求队列长度为Integer.MAX\_VALUE，可能会堆积大量请求，从而导致OOM。而是通过ThreadPoolExecutor的方式实现

## 线程池实现原理（过程）

* 1. 在创建了线程池后，等待提交过来的任务请求
  2. 当调用execut方法添加一个任务请求时，线程池会做如下判断：

1. 当提交一个新的任务时，如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（需要获得全局锁）
2. 如果运行的线程等于或者大于corePoolSize，则判断队列是否已满
3. 如果队列未满，则将任务添加到队列中（BlockingQueue），如果队列已满，则判断线程池是否已满（正在运行的线程数量是否小于maximumPoolSize）
4. 如果线程池未满，则创建新的非核心线程来执行任务（需要获得全局锁），如果线程池已满（数量超出maximumPoolSize），则任务被拒绝，按照策略处理无法执行的任务
   1. 当一个线程完成任务时，它会从队列中取出下一个任务来执行
   2. 当一个非核心线程空闲时间超过设置的超时时间后，这个非核心线程会停止，并且线程池会收缩到corePoolSize大小

|  |
| --- |
|  |

## 线程调度

线程调度：线程只有在获取了CPU使用权之后才可以开始执行代码，获取CPU的过程就是线程调度过程

获取CPU的方法：

1. 调用sleep()方法休眠，让出CPU使用权，Thread.sleep(1000)
2. 通过setPriority(1)方法设置优先级，优先级默认为5
3. 通过setDaemon(true)方法设置守护线程
4. 未捕获异常处理器，捕获到异常就让出CPU使用权
5. 线程让步Thread.yield()方法
6. 使用join()方法

## 线程池如何调优，最大数目如何确认

如果是CPU密集型：线程池最大数目=CPU个数+1 即：

Int maxThread = Runtime.getRuntime().availableProcessors()+1

如果是IO密集型：由于IO密集型任务线程并不是一直执行任务，则应配置尽可能多的线程，

（1）线程池最大数目=CPU个数\*2 即：

Int maxThread = Runtime.getRuntime().availableProcessors()\*2

（2）参考公式CPU核数/(1-阻塞系数)。阻塞系数在0.8~0.9之间

比如8核CPU：8/(1-0.9)=80

线程池调优：

1. 核心线程Warm up优化：在线程启动的时候Warm up核心线程，从而达到新的任务（线程）过来时可以立马执行的效果
2. 定制工作线程：默认Executors.defaultThreadFactory将被使用，这个时候所有的线程属于同一个线程组 ，拥有同样的优先级和daemon（守护线程）状态，如果调用Executors.threadFactory我们可以配置线程的名字、线程的组合和deamon状态
3. 监控线程池，及时回收空闲时间大的线程

（4）合理选择队列

（5）合理选择线程池

## 18. 原子类atomic的原理及原子引用

### 18.1 实现原理：Unsafe类

Unsafe是CAS的核心类，用来实现CAS，Unsafe类存在于rt.jar包下面的sun.misc包中，是jdk自带的；

Unsafe类中所有方法都是native修饰的，也就是说Unsafe类中所有方法都直接调用操作系统底层资源执行相应任务。

|  |
| --- |
|  |

1. unsafe字段，atomicInteger源码里包含了Unsafe.getUnsafe()类的实例private static final Unsafe unsafe = new Unsafe();
2. valueOffset：表示该变量在内存中的偏移地址，因为Unsafe类就是根据内存偏移地址获取数据的
3. value：value是用volatile修饰，保证多线程之间内存可见性。表示当前对象代码的基本类型的值，AtomicInteger是int型的线程安全包装类

AtomicInteger中i++的实现：

|  |
| --- |
|  |

### 18.2 原子引用（AtomicRefrence）

|  |
| --- |
|  |

## 7大阻塞队列

### 19.1 阻塞队列介绍

当阻塞队列为空时，从队列中获取元素的操作将会被阻塞

当阻塞队列满时，想队列中添加元素的操作将会被阻塞

|  |
| --- |
|  |

都是Collection🡪Queue🡪BlockQueue的实现

（1）ArrayBlockingQueue：由数组结构组成的有界阻塞队列。接收到任务时，如果没有达到corePoolSize的值，则新建核心线程执行任务；如果达到了，则进入队列等待；如果队列已满，则新建非核心线程处理任务；如果总线程数到了maximumPool，且队列已满，则发生异常

（2）LinkedBlockingQueue：由链表结构组成的有界（大小默认为：Integer.MAX\_VALUE）阻塞队列。接收到任务时，如果当前线程数小于核心线程数，则新建核心线程处理任务；如果当前线程数大于等于核心线程数，则进入队列等待

（3）PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列

（4）DelayQueue：一个使用有限级队列实现的无界阻塞队列。接收到任务时，首先进入队列，只有达到了指定的延时时间，才会执行任务。队列内元素必须实现Delayed接口，意味着任务必须先实现Delayed接口

（5）SynchronousQueue：不存储元素的阻塞队列，即单个元素的队列。接收到任务时，会直接提交给线程处理，而不保留它

（6）LinkedTransferQueue：由链表结构组成的无界阻塞队列

（7）LinkedBlockingDeque：由链表结构组成的双阻塞队列

### 19.2 常用方法介绍及区别

（1） 向阻塞队列中添加元素add()/offer()/put()

一些队列又大小限制，如果想在一个满的队列中添加一个对象时，多出的对象就会被拒绝。

add方法插入对象时会抛出IllegalStateException：Queue full异常

offer方法会返回false，但是可以设置超时时间，相当于设置阻塞时间，在超时时间内会一直阻塞，当超过设置的时间后，返回false

put方法会一直阻塞，无返回值

（2）从空的阻塞队列中移除(获取)第一个元素remove()/poll()/take()

remove方法会抛java.util.NoSuchElementException异常

poll方法会返回null，但是可以设置超时时间，相当于设置阻塞时间，在超时时间内会一直阻塞，当超过设置的时间后，返回null

take方法会一直阻塞，无返回值

（3）从空的阻塞队列的头部查询元素peek()/element()

element方法会抛出java.util.NoSuchElementException异常

peek方法会返回null

## 20. condition接口

Condition接口定义了等待/通知两种类型的方法，Condition对象是需要关联Lock对象的，经调用lock对象的newCondition()对象创建而来，也就是说Condition的实现是依赖Lock对象的

通过condition对象对线程阻塞和唤醒

|  |
| --- |
|  |

备注：Java中有两种等待通知机制

1. wait/notify (notifyAll)：在使用之前获取对象锁。调用方式（直接调用）：Thread.wait()
2. await/signal (signalAll)：通过lock.lock()获取对象锁，调用方式（通过lock对象调用）：lock对象的newCondition()创建condition对象，condition.await()

## 多线程中单例模式的使用

单线程中单例模式的使用

1. 私有的构造方法
2. 创建类的唯一实例
3. 提供一个用于获取实例的方法

|  |
| --- |
|  |

多线程中单例模式的使用（DCL模式，Double Check Lock,使用双重判定并加锁），锁加在方法内部，这样在调用的时候无需加锁，只有在创建对象的时候再加锁，这样做的好处是，因为对象只会创建一次，所以并不会影响性能，也可以使用synchronized关键字（不建议）

|  |
| --- |
|  |

但是DCL并不一定保证线程安全，因为有指令重排序的存在，可以使用volatile关键字解决，多线程下单例模式使用：DCL+锁+volatile

|  |
| --- |
| 解决：创建类的唯一实例时添加volatile关键字  Private static volatile Singleton singleton = null; |

# Spring

## FactoryBean和BeanFactory

### 1.1 FactoryBean

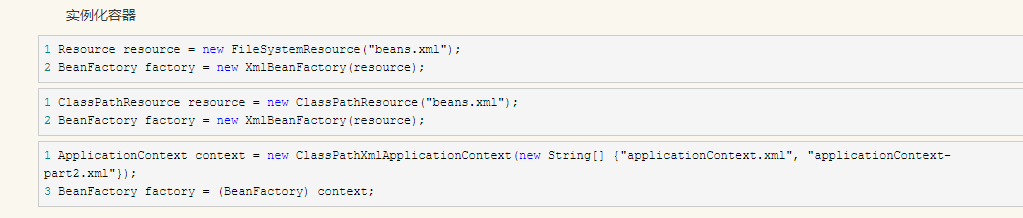
factoryBean以bean结尾，它表示一个Bean，不同于普通Bean的是：它是实现了FactoryBean<T>接口的Bean，根据Bean的ID从FactoryBean中获取的实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身，如果要获取FactoryBean本身，请在id前面加一个&符号来获取



### 1.2 BeanFactory

BeanFactory以factory结尾，表示它是一个工厂类（接口），用于管理bean的一个工厂。在spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖

Spring为我们提供了许多易于使用的BeanFactory实现，XmlBeanFactory就是其中的一个，该实现将以xml方式描述组成应用的对象以及对象之间的依赖关系。XMLBeanFactory类将持有此xml配置元数据



## Spring有哪些模块

|  |
| --- |
|  |

1. 核心容器（spring core）

核心容器提供spring框架的基本功能，spring以bean的方式组织和管理java应用中的各个组件及其关系。Spring使用beanFactory来产生和管理bean，它是工厂模式的实现。beanFactory使用控制反转（IOC）模式将应用的配置和依赖性规范与实际的应用程序代码分开

1. 应用上下文（spring context）

Spring上下文是一个配置文件，向spring框架提供上下文信息

1. 面向切面编程（spring aop）

通过配置管理特性，spring aop模块直接将面向切面的编程继承到spring框架中，使spring框架管理的任何对象支持aop。Spring aop模块为基于spring的应用程序中的对象提供了事务管理服务。通过使用spring aop，不用依赖EJB组件，就可以将声明式事务管理集成到应用程序中

1. Jdbc和dao模块（spring dao）

可用改结构来管理异常处理

1. 对象实体映射（spring orm）
2. Web模块（spring web）
3. Mvc模块（spring mvc）

## Spring的加载过程

1. 首先，对于一个web应用，其部署在web容器中，web容器提供一个全局的上下文ServletContext，其为后面的spring IOC容器提供宿主环境
2. 其次，在web.xml中会提供有contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到此事件，其初始化方法contextInitialized方法会被调用，在这个方法中，spring会初始化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext，这是一个接口类，确切的说，其实现的是一个XmlWebApplicationContext。这个就是SpringIOC容器，其对应的bean定义的配置由web.xml中的context-param标签指定
3. 最后，contextLoaderListener初始化完毕后，开始初始化web.xml中的servlet，springMVC的核心控制器DispatcherServlet，用于转发、匹配、处理每个servlet请求。

## Spring的核心组件

1. Bean

Bean的定义：spring配置文件中定义的<bean/>节点的所有信息

Bean的创建：利用BeanFactory创建bean对象，xmlBeanFactory

Bean的解析：Bean的解析主要就是对spring配置文件的解析（又回到了spring的加载过程）

1. Context

给spring提供一个运行时环境，用于保存各个对象的状态，applicationContext是context的顶级父类，他除了能表示一个应用环境外，还继承了5个接口，用于扩展context功能

1. Core

定义了资源的访问方式，把所有的资源都抽象成一个接口。

Resource接口封装了各种可能的资源接口

## Spring Bean的生命周期

1. 实例化一个Bean
2. IOC注入：按照spring上下文对实例化的bean进行配置
3. 实现BeanNameAware接口，实现它的setBeanName(string)方法，此处传递的是spring配置文件中的id值
4. 实现BeanFactoryAware接口，实现setBeanFactory(BeanFactory)方法，此处传递的是spring工厂本身
5. 实现ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入spring上下文（这个方式也可以实现步骤4的内容，但比4更好，因为ApplicationContext是BeanFactory的子接口，有更多的实现方法）
6. 执行BeanPostProcessor的postProcessBeforeInitialization(Object obj,String str)方法,BeanPostProcessor经常被用作是Bean内容的更改
7. 调用spring配置文件中<bean>的init-method属性指定的初始化方法
8. 执行BeanPostProcessor的postProcessAfterInitialization(Object obj,String str)方法
9. 容器初始化成功，执行正常调用后，然后容器销毁
10. 实现DiposibleBean接口的destory方法
11. 调用spring配置文件中的<bean>的destory-method属性指定的初始化方法

## Spring IOC的理解，其初始化过程

原理：IOC（控制反转）是Spring的核心，context作为springIOC的容器，由spring来控制对象的生命周期和对象间的依赖关系，以前我们想要调用一个对象自己通过new来实现，通过Spring IOC所有对象的创建、销毁等都由spring来控制，但是我们在一个对象中调用另一个对象，这一点是通过DI（依赖注入：反射）来实现的，使用最频繁：通过@autowrie或者@Resource在controller中注入service或者在service中注入dao，在使用时不需要自己创建，spring会帮我们实现new的操作。eg：以前对象A要操作数据库，自己编写代码来获取connection连接，现在我们可以通过spring来获取connection连接，在系统运行时，spring会在适当的时机创建一个connection连接以供使用。

实现过程：我们每次使用spring框架都会配置xml文件，这个xml配置了bean的id和class，spring中bean默认是单例模式，通过bean的class引用反射机制可以创建这个实例。因此，spring框架通过反射替我们创建好了实例并且替我们维护他们。

## Spring创建对象的方式（三种）：

3.1调用默认的构造方法

3.2 spring调用工厂方法，但真正创建对象还是程序员

3.3 实例工厂方法

## 实现Spring IOC需要注意的问题

Spring的做法是通过反射加载进来，当使用@Autowried或者@Resources注入的时候，要在对应的模块中加入对应的注解

实体类：@Entity

控制层：@controller

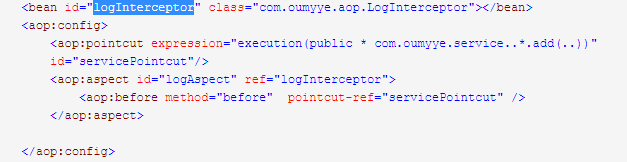
业务层：@service

数据连接层：@repository

## Spring AOP实现原理

XML配置实现AOP事务控制





原理：AOP（面向切面编程）可以说是OOP（面向对象编程）的补充和完善，OOP引入封装、继承、多态。AOP做一些事务处理、日志记录、权限控制等，eg:声明式事务管理的时候，我们在配置文件里面配置以update、save等开头的方法，当我们在service检测到这些方法被调用时，我们就先进性开启事务然后执行代码逻辑，最终关闭事务。

AOP相关概念：

Aspect（切面）:表示该类实现了增强，所谓切面，通常是指一个类，里面可以定义切入点和通知。简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或者责任封装起来，便于减少系统的重复代码，降低模块之间的耦合性。

Id：切面名 ref：方法所在的类 method：方法名

<aop:aspect id="logAspect" ref="logInterceptor">

<aop:before method="before" pointcut-ref="servicePointcut" />

</aop:aspect>

Pointcut（切入点）：声明一个切入点，expresstion表示满足这个方法就回去进行切面操作

<aop:pointcut expression="execution(public \* com.oumyye.service..\*.add(..))"

id="servicePointcut"/>

Joinpoint（连接点）：程序执行过程中明确的点，如方法的调用或特定的异常被抛出

Advice（通知/增强方式）：aop在特定的切入点上执行的增强处理，有before（执行前增强）、after（最终增强）、around（环绕增强），afterException（抛出异常时增强）、afterReturning（执行后增强）

实现AOP：（1）采用动态代理：jdk动态代理和cglib动态代理

1. 采用静态代理（静态织入）

实现方式：实现接口、注解、xml配置文件（aop标签）

spring对aop的支持：

Spring中aop的代理由spring的ioc容器负责生成、管理，其依赖关系也由ioc容器负责管理。因此，aop代理可以直接使用容器中的其它bean实例作为目标，这种关系可由ioc容器的依赖注入提供。Spring创建代理的规则为：

1. 默认使用jdk动态代理来创建aop代理，这样可以为任何接口实例创建代理
2. 当需要代理的类不是代理接口的时候，spring会切换为使用cglib动态代理，也可以强制使用cglib

### spring aop事务处理的使用

见7.2

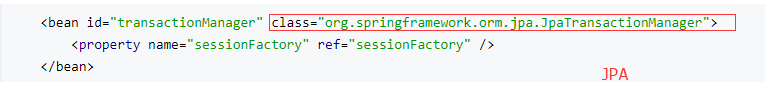
## Spring如何管理事务，事务管理机制？

事务：要么同时成功，要么同时失败，eg：取钱

事务四个特性：A（原子性）C（一致性）I（隔离性）D（持久性）

事务原本是数据库中的概念，在Dao层。但一般情况下，需要将事务提升到业务层，即Service层。这样做是为了能够使用事务的特性来管理具体的业务。

Spring并不直接管理事务，而是提供了多种事务管理器，Spring管理器的接口是：org.springframework.transaction.PlatformTransactionManager，主要用于完成事务的提交、回滚及获取事务的状态信息，通过这个接口，Spring为各个平台：JDBC、ibatis、Hibernate、JPA等提供了对应的事务管理器



在spring中通常使用以下四种方式来实现对事务的处理：

1. 编程式：使用编程式事务管理（一般不用）
2. 声明式：使用spring的事务代理工厂管理事务
3. 声明式：使用spring的事务注解管理事务，@transaction
4. 声明式：使用aspectJ的aop配置管理事务

### 使用注解@Transactional配置事务管理

1. 在事务管理的service实现类上面添加@Transactional注解

|  |
| --- |
|  |

1. 在要进行事务管理的方法上面添加注解

|  |
| --- |
|  |

1. 在spring配置文件中（applicationcontext.xml）指定驱动：

|  |
| --- |
|  |

### @Transactional可以作用于接口、接口方法、类及类方法上。当作用于类上时，该类的所有public方法都将具有该类型的事务属性，同时我们也可以在方法上使用注解来覆盖类级别的定义。但是不建议在接口和接口方法上使用该注解，因为这只有在基于使用接口代理时才会生效。而且该注解只被应用于public方法上，因为类内部方法调用不会引起事务行为

### 7.2 使用spring aop配置事务管理

1. 配置事务管理类

在spring配置文件中配置数据源（datasource）、事务管理器，不同的框架使用不同的数据源，以下是mybatis和jdbc的

|  |
| --- |
|  |

1. 配置事务属性

|  |
| --- |
|  |

不要使用get和set关键字，防止和属性的getter和setter发生混淆

1. 配置事务的AOP切入点

|  |
| --- |
|  |

通过以上三个步骤设置好声明式事务后，当service中的业务方法被调用之前，spring会获取事务对象并开启事务。并使用try-catch-finally来处理异常。业务方法执行成功后则会提交事务。默认抛异常或者发生error时事务回滚。

在tx:method中配置了rollback\_for中配置的exception，这个是运行时异常才会回滚，其他异常不回滚

## Spring的事务传播行为及隔离级别

7种传播行为：

|  |  |
| --- | --- |
| **传播行为** | **含义** |
| PROPAGATION\_REQUIRED | 表示当前方法必须运行在事务中，如果事务存在，则在该事务中运行，如果不存在，则启动一个新的事务 |
| PROPAGATION\_SUPPORTS | 表示当前方法不需要事务上下文，如果事务存在，则在该事务中运行，如果不存在，不在事务中运行 |
| PROPAGATION\_MANDATORY | 表示当前方法必须在事务中运行，如果事务不存在，则会抛异常 |
| PROPAGATION\_REQUIRED\_NEW | 表示当前方法必须运行在自己的事务中，一个新的事务将被启动，如果存在当前事务，在该方法执行期间，当前事务会被挂起 |
| PROPAGATION\_NOT\_SUPPORTED | 表示当前方法不应该运行在事务中，如果存在当前事务，在该方法执行期间，当前事务会被挂起 |
| PROPAGATION\_NEVER | 表示当前方法不应该运行在事务上下文中，如果当前正有一个事务在运行，则会抛出异常 |
| PROPAGATION\_NESTED | 如果当前已经存在一个事务，则该方法会在嵌套事务中运行，嵌套事务可以独立于当前事务进行单独的提交或回滚，如果当前事务不存在，则与PROPAGATION\_REQUIRED的传播行为一样 |

5种隔离级别：ISOLATION\_DEFAULT:底层数据库默认隔离级别

ISOLATION\_READ\_UNCOMMITED（未提交读）

ISOLATION\_READ\_COMMITED（提交读）

ISOLATION\_REPEATABLE\_READ（可重复读）

ISOLATION\_SERIALIZABLE（序列化）

## Spring中用到了哪些设计模式

（1）单例模式：Spring中bean默认是单例模式

（2）前端控制器：Spring用前端控制器DisatcherServlet请求转发

（3）工厂方法模式：BeanFactory创建对象的实例

（4）代理模式：Spring AOP的实现中用到了JDK动态代理

（5）迭代器模式：Iterable接口和Iterator接口

（6）模板方法模式：jdbcTemplate

（7）策略模式：spring中事务管理机制

## SpringMVC工作原理

核心：控制反转（IOC）和面向切面（AOP）

1. 客户端发送请求到DispatcherServlet（所有请求都由DispatcherServlet统一分发，核心控制器，截取URL请求）
2. 由DispatcherServlet控制器查询HandlerMapping，找到处理请求的controller（初始化DispatcherServlet上下文对应的WebApplicationContext，与controller建立关联）
3. DispatcherServlet将请求发送到对应的controller
4. controller调用对应的业务逻辑，返回ModelAndView或者Model
5. DispatcherServlet查询ViewResoler视图解析器，找到ModelAndVIew或者Model指定的视图
6. 由ModelAndView或者Model相应到前端jsp页面，给用户展示

Xml中配置SpringMVC:



## Spring依赖注入方式

依赖注入：对于spring配置一个bean时，如果需要给bean提供一些初始化参数，则需要通过依赖注入方式，所谓依赖注入就是通过spring将bean所需要的一些参数传递到bean实例对象的过程

#### 属性注入：setter注入

对象中的属性设置get和set方法，如果类中不定义构造函数，默认会有一个无参构造函数，如果定义了构造函数，将不会默认生成无参构造函数，需要提供一个默认的无参构造函数，否则setter注入会异常

applicationContext.xml中定义属性注入

|  |
| --- |
|  |

测试：

|  |
| --- |
|  |

* 1. 构造器注入

必须要有有参构造函数

1. 按类型匹配入参

|  |
| --- |
|  |

1. 按索引匹配入参

|  |
| --- |
|  |

1. 联合使用索引和类型匹配入参

|  |
| --- |
|  |

* 1. Filed注入（注解）

使用注解方式注入可以注入对象

@Resource(name="commonDao")：默认按名称装配，如果没有找到该名称，则会按照类型去寻找

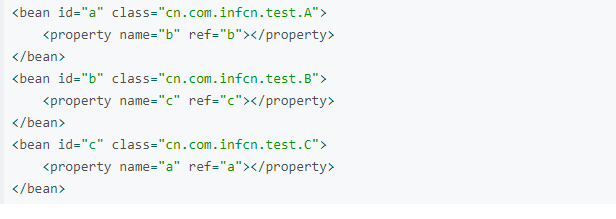
@Autowired：默认按类型装配

## Spring循环依赖注入

1. 构造器循环依赖注入（Spring会将每一个正在创建的bean标识符放在一个“当前正在创建的Bean池”中，并且在创建过程中一直存在，eg：A中创建B，B中创建C，C中创建A，在创建的过程中发现自己已经在池中，就会造成循环，抛出异常BeanCurrentlyCreationException，创建完毕将从池中清除掉）



1. 属性注入：setter方法循环注入（单例和非单例，Spring先是构造实例化Bean对象，此时Spring会将这个实例化结束的对象放到一个Map中，并且Spring提供了获取这个未设置属性的实例化对象引用的方法，当Spring实例化了ABC之后，紧接着设置对象的属性，此时A依赖B,就会去Map中取出存在里面的单例B对象，不会循环）



## SpringJPA

SpringJPA类似于Hibernate

在使用的时候我们需要继承JpaRepository<对应表的实体类，主键类型>

## SpringBoot和SpringCloud的使用

### SpringBoot

#### SpringBoot搭建步骤

1. 下载maven服务、配置maven环境变量
2. 在eclipse中配置maven服务，指定 settings.xml文件位置，并修改settings.xml文件，修改maven仓库地址和镜像文件
3. 在eclipse中安装Spring Tools Suite
4. 创建项目并在applications.properties中添加配置文件
5. 启动项目：右键项目点击debug as中的Spring Boot App

注意：启动项目时要将application放到最外层，否则会报404错

#### SpringBoot中yml和yaml的使用

SpringBoot中yml/yaml的作用与application.properties作用一样，都是用来存放配置文件，只是两者的写法不同

个人比较倾向于application.properties

* + 1. SpringBoot中事务处理

一般采用注解@Transactional

* + 1. SpringBoot中健康监控

导入依赖，访问localhost:8080/health

### 12.2 SpringCloud

微服务框架，一般和SpringBoot结合使用

#### 12.2.1 服务发现与注册（Eureka）

服务端配置：

1. 添加依赖和配置文件
2. 启动类里面添加@EnableEurekaServer注解

客户端配置：

1. 添加依赖和配置文件
2. 启动类里面添加@EnableEurekaClient注解

启动服务端和客户端，会自动将服务注册到服务器

#### 12.2.2 负载均衡（Ribbon）

多个服务在Eureka服务器里面的应用名称必须相同

1. 修改配置文件和启动类
2. 在controller类里面导入restTemplate模板

#### 12.2.3 接口调用（Feign）

服务端配置：

1. 添加依赖和配置文件
2. 启动类里面添加注解
3. Controller类里面编写接口

客户端实现：

1. 添加依赖和配置文件

#### 12.2.4 路由（Zuul）

#### 12.2.5 熔断机制（Hystrix）

当请求出现问题时，不进行请求，去请求其他方法，阻止了异常的发生

# Tomcat优化

## Tomcat优化（内存、并发、缓存）

### Tomcat内存优化

Tomcat内存优化主要是对Tomcat启动参数优化，我们可以在Tomcat的启动脚本catalina.sh中设置JAVA\_OPTS参数：

JAVA\_OPTS="-Dfile.encoding=UTF-8 -server -Xms1024m -Xmx1024m -XX:NewSize=512m -XX:MaxNewSize=512m -XX:PermSize=256m -XX:MaxPermSize=256m -XX:NewRatio=2 -XX:MaxTenuringThreshold=50 -XX:+DisableExplicitGC"

JAVA\_OPTS参数说明：

-server：启动jdk的server版本

-Xms：Java虚拟机初始化时的最小内存

-Xmx：Java虚拟机初始化时的最大内存

-XX:NewSize：设置年轻代大小

-XX:MaxNewSize：设置最大的年轻代大小

-XX:PermSize：设置永久代大小

-XX:MaxPermSize：设置永久代最大大小

-XX:NewRatio=4：设置Young区（包括Eden和2个survior）和终身区（除去永久代）的比值，设置为4，则说明年轻代与终身代的比值为1:4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=32意味着eden和一个survivor的比值是32：1，这样一个Survivor就占Young区的1/34.

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄，默认为：15。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

-XX:+DisableExplicitGC这个将会忽略手动调用GC的代码使得 System.gc()的调用就会变成一个空调用，完全不会触发任何GC

### Tomcat并发优化

Tomcat并发（线程池）参数优化：修改Tomcat中server.xml文件参数，比如修改最小处理线程数等，用于提高系统处理性能

修改参数配置前：打开这个注释

|  |
| --- |
|  |

|  |
| --- |
|  |

修改参数配置后及参数说明：

maxThreads：客户请求最大线程数

minSpareThreads：Tomcat初始化时创建的socket线程数

maxSpareThreads：Tomcat连接器的最大空闲socket线程数

minProcessors：服务器创建时的最小处理线程数

maxProcessors：服务器同时最大处理线程数

enableLookups：若设为true，则支持域名解析，可把ip地址解析为主机名

redirectPort：在需要基于安全通道的场合，把客户请求转发到基于SSL的redirectPort端口

AcceptCount:监听端口队列最大数，满了之后客户请求会被拒绝（不能小于maxSpareThreads）

connectionTimeout：设置客户连接超时的时间，如果为-1，表示不限制建立客户连接的时间

threadPriority：

|  |
| --- |
|  |

线程池配置：

|  |
| --- |
|  |

### Tomcat缓存优化

修改Tomcat中的server.xml配置文件

修改前参数配置：

|  |
| --- |
|  |

修改后参数配置及参数说明：

compression：打开压缩功能

compressionMinSize：启动压缩的输出内容大小，默认2KB

compressableMimeType：压缩类型

|  |
| --- |
|  |

### 1.4 Tomcat3种运行模式

（1）bio：同步并阻塞，默认运行模式，一个客户端连接创建一个线程，如果客户端连接不做任何操作会在成不必要的线程开销，性能非常低下，没有经过任何优化处理和支持。可以通过线程池优化。

适用于连接数比较小的架构，jdk1.4之前

1. nio(new I/O)：同步非阻塞，一个请求创建一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才会启动线程处理。是Java SE 1.4及后续版本提供的一种新的I/O操作方式(即java.io及其子包)，java.nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java AP，因此nio也被看成是no-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能

适用于连接数比较多且连接比较短的架构

1. aio：异步非阻塞，一个有效请求创建一个线程，但是可以从操作系统级别来解决异步的I/O问题，大幅度的提高性能

适用于连接数较多且连接比较长的架构

启动nio模式：修改server.xml配置文件中的connector节点中protocol的值为：org.apache.coyote.http11.Http11NioProtocol

|  |
| --- |
|  |

# Redis缓存

## Redis有哪些数据类型及常用命令

### String（字符串）

|  |  |  |
| --- | --- | --- |
| String常用命令 | | |
| 含义 | 命令 | 语法 |
| 存值/取值/删除 | set/get/del | set key value / get key / del key |
| 自增/减 | Incr/decr | Incr key / decr key |
| 获取字符串长度 | strlen | strlen key |
| 获取多个键值 | mget | mget key1 key2 |
| 设置多个键值 | mset | mset key1 value1 key2 value2 |
| 设置过期时间 | expire | expire key 100 |
| 查看生存时间 | ttl | ttl key |

### List（集合）

|  |  |  |
| --- | --- | --- |
| List常用命令 | | |
| 含义 | 命令 | 语法 |
| 存值 | LPUSH (插入表头) RPUSH (插入表尾) | LPUSH key value 返回列表长度  RPUSH key value 返回列表长度 |
| 移除 | LPOP（移除头元素）  RPOP（移除尾元素） | LPOP key 并返回被移除的头元素的值  RPOP key 并返回被移除的尾元素的值 |
| 获取集合长度 | LLEN | LLEN key |
| 获取指定区间值 | LRANGE key start end | LRANGE key 0 0/1 返回一/两个值 |
| 获取指定位置值 | LINDEX key index | LINDEX key 0 返回第一个值 |
| 设置指定位置值 | LSET key index value | LSET key 1 value 给第二个位置设 |

List数据类型可以实现消息队列，LPUSH作为消息生产者，LPOP作为消息消费者

### Set（集合）

|  |  |  |
| --- | --- | --- |
| Set常用命令 | | |
| 含义 | 命令 | 语法 |
| 存值 | sadd | sadd key value1 value2 对key添加一个或多个value |
| 移除 | srem | srem key value1 value2 移除集合中一个或多个元素 |
| 移动 | smove | smove value key2 key1 将value从key1移到key2 |
| 获取元素个数 | scard | scard key 返回元素个数 |
| 获取集合key值 | smembers | smembers key |

### zSet（有序集合）

|  |  |  |
| --- | --- | --- |
| zSet常用命令 | | |
| 含义 | 命令 | 语法 |
| 存值 | zadd | zadd key value1 value2 对key添加一个或多个value |
| 移除 | zrem | zrem key value1 value2 移除集合中一个或多个元素 |
| 返回元素排名 | zrank | Zrank key value1 返回value1在集合中排名，值从小到大排列 |
| 获取元素个数 | zcard | zcard key 返回元素个数 |
| 获取集合key值 | smembers | smembers key |

### 1.5 Hash（散列）

|  |  |  |
| --- | --- | --- |
| Hash常用命令 | | |
| 含义 | 命令 | 语法 |
| 存值/取值 | hset  hget | hset hkey key value  hget hkey key |
| 移除 | hdel | hdel hkey key |
| 获取元素个数 | hlen | hlen hkey |
| 获取多个key值 | hmget | hmget hkey key1 key2 |
| 给多个可以赋值 | hmset | hmset hkey key1 value1 key2 value2 |
| 获取所有key | hkeys | hkeys hkey |
| key是否在hkey | hexists | hexists hkey key 返回1/0 |

## Redis底层实现

### 2.1 Redis底层8种数据结构

|  |  |
| --- | --- |
| 编码常量 | 编码所对应底层数据结构 |
| REDIS\_ENCODING\_INT | Long类型的整数 |
| REDIS\_ENCODING\_EMBSTR | Embstr编码的简单动态字符串 |
| REDIS\_ENCODING\_RAW | 简单动态字符串 |
| REDIS\_ENCODING\_HT | 字典 |
| REDIS\_ENCODING\_LINKEDLIST | 双端链表 |
| REDIS\_ENCODING\_ZIPLIST | 压缩列表 |
| REDIS\_ENCODING\_INTSET | 整数集合 |
| REDIS\_ENCODING\_SKIPLIST | 跳跃表和字典 |

### 2.2 Redis中5种数据类型对应的底层实现结构

#### 2.2.1 String（字符串）

String底层对象的编码可以是int、raw、embstr

1. 如果一个字符串内容可以转为Long，那么该字符串就会被转为Long类型，对象的ptr就会指向该Long，并且对象类型也用int表示
2. Embstr是redis3.0后新加的，如果字符串对象的长度小于39字节，就用embstr，否则就用传统的raw对象

#### 2.2.2 List（List列表）

List对象的底层编码可以是ziplist或者linkedlist

1. ziplist是一种压缩列表，好处是节省内存空间，因为它存储的内容都是在连续的内存区域当中的，当列表对象和每个元素都不大的时候采用ziplist，数量过大不适用，因为需要保证存储内容在内存中的连续性，每次插入都会重新realloc，插入的复杂度是O(N)
2. Linkedlist是一种双向链表，节点中存有pre和next两个指针，还有节点相关信息，每当增加一个node的时候，就需要重新malloc

#### 2.2.3 Hash（哈希）

Hash磁层实现的结构对象可以是ziplist或者hashtable

1. ziplist见List中的ziplist
2. HashTable是由dict结构来实现的

|  |
| --- |
|  |

#### 2.2.4 Set（Set集合）

Set集合底层数据结构对象可以是intset或者hashTable

1. intset是一个有序集合，支持升级操作，所有数据都升级，如从int16\_t升级到int32\_t，不支持降级操作
2. hashTable见Hash中的hashTable

#### 2.2.5 zSet（zSet有序集合）

zSet对象底层数据结构可以是ziplist或者skiplist和dict的结合

1. ziplist见List中的ziplist
2. Skiplist是一种跳跃表，它实现了有序集合中的快速查找

## Redis缓存穿透，缓存雪崩

### 缓存穿透

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需要从数据库中查，查询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数据库去查询，造成缓存穿透。

解决办法：

1. 布隆过滤：对所有可能查询的参数以hash形式存储，先控制层进行校验，不符合则丢弃。最常见的是采用布隆过滤器，将所有可能存在的数据hash到一个足够大的bitmap中，一个一定不存在数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力
2. 当我们从数据库找不到的时候，我们将这个空对象也就行缓存，下次再请求时，不用去请求数据库，将过期时间尽量设短，最长不超过5分钟（否则如果长时间不过期的话缓存中就会更多的键，这时需要足够大的内存），存在的问题：可能会造成缓存中和存储系统中数据不一致问题（在缓存空对象未过期这段时间内，存储系统中对这个key设置了值，这时并没有及时更新到缓存中，造成从缓存中取出来的值仍未空，解决：可以在对数据库做了增删改之后清除其缓存，查询时从数据库里查再将其缓存到缓存里，或者做一个消息系统，当数据库更改之后通过消息队列修改其缓存中的内容）

### 缓存雪崩

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有查询都落到数据库上，造成了缓存雪崩。

解决办法：

1. 做二级缓存，当一级缓存失效后，从二级缓存中去查
2. 数据预热：可以通过reload机制，预先去更新缓存，在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间（随机值），让缓存失效的时间点尽量均匀

加锁：当缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量，当缓存失效后，不是立即去查询数据库，先使用缓存工具将某些成功返回值的数据做缓存

（3）实现redis高可用，主从架构或redis集群，避免redis挂掉

## Redis持久化方式，优缺点及如何实现

### snapshotting（RDB快照）

快照是默认的持久化方式。这种方式就是将内存中数据以快照的方式写入二进制文件中，默认的文件名为dump.rdb。可以通过配置设置自动做快照的持久化方式。我们可以设置在n秒内超过m个key被修改就自动做快照

|  |
| --- |
|  |
|  |

快照保存过程：

1. redis调用fork()函数，产生一个子进程，现在有了子进程和父进程
2. 父进程继续处理clien请求，子进程负责将内存内容写入临时文件中。由于os的写时复制机制（copy on write）父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面，所以子进程的地址空间内的数据是fork时刻整个数据库的一个快照
3. 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出

缺点：

1. save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有的client请求，这种方式会阻塞所有client请求
2. 每次快照都是将快照数据完整写入到磁盘一次，并不是增量保存，所以数据量大并且写的次数多必然会造成大量的磁盘io操作
3. 由于快照方式是在一定间隔时间做一次的，所以如果redis意外down掉的话，会丢失最后一次的数据

优点：

1. 方便备份
2. 恢复大数据集比AOF快

RDB对过期key的处理：过期key对RDB没有影响

从内存库持久化数据到RDB文件之前会检查key是否过期，过期key不进入RDB文件

从RDB文件恢复数据到内存库之前会对key进行过期检查，如果过期，不导入数据库

### Append-only file（aof）--推荐使用

Aof方式比快照有更好的持久化性，是由于在使用aof持久化方式时，redis会将每一个收到的写命令都通过write函数追加到文件中，当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库内容，由于os会在内核中缓存write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也有可能会丢失部分修改。不过我们可以通过配置文件告诉redis通过fsync函数强制os写入到磁盘的时机

|  |
| --- |
|  |

保存过程：

1. Redis调用fork，现在有父子两个进程
2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令
3. 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题
4. 当子进程把快照内容写入（以命令方式）临时文件中后，子进程发信号通知父进程，然后父进程把缓存的写命令也写入到临时文件
5. 现在父进程可以使用临时文件替换老的aof文件，并重命名，候命收到的写命令也开始往新的aof文件中追加

AOF对过期key的处理：过期key对AOF没有任何影响

从内存库持久化数据到AOF文件时如果key过期，没有被删除，此时执行持久化操作（该key不会写入AOF文件，因为没有发生修改命令），如果key被删除，会向AOF文件中添加一条del命令（在将来恢复到内存数据库时该过期key被删除）

AOF重写时会判断key是否过期，如果过期不会重写到AOF文件

## Redis缓存失效策略和过期策略

设置过期时间：（1）expire key time 最常用，支持5种数据类型

（2）setex key time value 字符串独有

3种过期策略：

1. 定时删除：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，删除key

优点：保证内存被尽快释放

缺点：若过期key很多，删除这些过期key会占用很多的cpu时间，若cpu时间紧张的话，cpu不能把时间用来处理有效事件，而是用来删除过期key。

定时器的创建耗时，若为每一个设置过期时间的key创建一个定时器，将会产生大量的定时器，影响性能

1. 惰性删除：key过期的时候不删除，每次从数据库获取key的时候去检查是否过期，若过期，则删除

优点：对cpu时间占用比较少，删除操作只发生在从数据库读取key的时候

缺点：若大量key超出过期时间，很长一段时间没有被获取过，可能发生内存泄露

1. 定期删除：每隔一段时间执行一次删除过期key的操作

优点：cpu时间方面比定时删除好

定期删除会防止惰性删除的内存泄露问题

缺点：内存方面不如定时删除

cpu时间方面不如惰性删除

Redis使用的过期策略：惰性删除+定期删除

## Redis数据淘汰策略(用于释放内存)

Redis可以通过Maxmemory指令配置Redis的数据集使用指定量的内存。设置Maxmemory=0，则表示无限制，超过最大内存，则会使redis崩溃，所以一定要设置

server.Maxmemory=100MB

当内存使用达到maxmemory的极限值时，需要使用某种淘汰算法来决定清理掉哪些数据，以保证新数据的存入

常用的淘汰算法：

1. FIFO (First In First Out)：先进先出，判断被存储的时间，离目前最远的数据优先被淘汰
2. LRU (Least Recently Used)：最近最少使用，判断最近被使用的时间，目前最远的数据优先被淘汰
3. LRU (Least Frequently Used)：最不经常使用，在一段时间内，数据被使用次数最少的，优先被淘汰

Redis提供的淘汰策略：

1. no-eviction：禁止淘汰数据
2. allkeys-random：从数据集随机选择数据淘汰
3. allkeys-lru：从数据集挑选最近最少使用的数据淘汰
4. volatile-random：从已设置过期时间的数据集中随机淘汰数据
5. volatile-ttl：从已设置过期时间的数据集选择将要过期的数据淘汰
6. volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰

## Redis集群、高可用原理

Redis集群搭建的方式有多种，例如使用zookeeper等，但在Redis3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态，每个节点都和其它所有节点连接

Redis集群结构特点：

1. 所有Redis节点彼此互联（PING-PONG机制），内部使用二进制协议优化传输速度和宽带
2. 节点的file是通过集群中超过半数的节点检测失效才生效
3. 客户端与Redis节点直连，客户端不需要连接集群中所有节点，连接集群中任何一个可用节点即可
4. Redis-Cluster把所有的物理节点映射到[0-16383]槽（slot）上（不一定是平均分配）
5. Redis集群预先分配好16384个槽，当需要在Redis集群中放置一个key-value时，根据哈希槽（hash slot）的方式决定将key放到哪个槽中

Redis-Cluster节点分配：

有几个节点分配几个槽，如3个节点ABC

节点A覆盖0-5460

节点B覆盖5461-10922

节点C覆盖10923-16383

如果存入一个值，根据Redis-Cluster哈希槽算法：CRC16(‘KEY’)384=6782，呢么就会把这个key分配到B节点上了，连接ABC任意一个节点都可以获取到其他节点上的数据，同理，如果有4个节点的话将槽分成4个

Redis-Cluster主从模式：

Redis-Cluster为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据获取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会由这些从节点选取一个主节点，从而保证集群不会挂掉

如上面ABC三个节点，如果不给其分配从节点的话，其中任何一个节点挂掉，集群将挂掉；如果一个主节点和其所有从节点都挂掉的话，那么这个集群也就挂掉不能提供服务了；如果主节点中超过一半的主节点挂掉的话，无论其是否有从节点，集群将挂掉

## Redis缓存分片

分片（partitioning）就是将你的数据拆分到多个Redis实例的过程，这样每个实例将只包含所有键的子集

范围分片：假如有ABC三个Redis实例，那么将用户0-100，放到A里面，这就是范围分片

# 分布式相关

## 对分布式系统的理解

分布式系统就是将服务分为服务提供者和服务消费者，将一个大的工程分成了多个小的工程，每个小的工程可以独立运行并且完成一个特定的模块，每个工程可以使用不同的语言进行开发，易于开发和维护，解决了传统项目单层架构存在的问题，但是分布式系统存在接口成本高和运维成本高的缺点

原则：

1. 单一职责原则：高内聚低耦合（最大程度解耦）
2. 服务自治原则：各个服务之间互不影响
3. 轻量级通信原则：跨平台、跨语言
4. 接口明确原则：接口必须明确

## Dubbo的底层实现原理

解析服务：

1. 基于dubbo.jar内的Meta-inf/spring.handlers配置，spring在遇到Dubbo名称空间时，会调用DubboNameSpaceHandler类
2. 所有的dubbo标签，都统一用DubboBeanDefenitionParser进行解析，基于一对一属性映射，将xml标签解析为Bean对象
3. 在ServiceConfig.export或者ReferenceConfig.get初始化时，将Bean对象转化为URL格式，将所有Bean属性转化为URL参数
4. 然后将URL参数传给protocol扩展点，基于扩展点的Adaptive机制，根据URL的协议头，进行不同协议的服务暴露和引用

暴露服务：

1. 只暴露服务端口：

在没有使用注册中心的情况，这种情况一般使用在开发环境中，服务的调用者和提供者在同一个IP上，只需要打开服务的端口即可，即ServiceConfig解析出来的URL格式为

Dubbo://service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptivr机制，通过URL的”dubbo://”协议头识别，直接调用DubboProtocol的export()方法，打开服务端口

（2）向注册中心暴露服务：

需要将服务的IP和端口一同暴露给服务中心

ServiceConfig解析出来的URL格式为

Registry://registry-host/com.alibaba.dubbo.registry.RegistryService?export=URL.encode(“dubbo://service-host/com.xxx.TxxService?version=1.0.0”)

基于扩展点的Adaptive机制，通过URL的”registry://”协议头识别，调用RegistryProtocol的export()方法，将export参数中的提供者URL先注册到注册中心，再重新传给Protocol扩展点进行暴露：

Dubbo://service-host/com.xxx.TxxService?version=1.0.0

引用服务；

1. 直接引用服务：

在没有注册中心，直连提供者的情况下ReferenceConfig解析出来的URL格式为：

Dubbo://service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过url的”dubbo://”协议头识别，直接调用DubboProtocol的refe方法，返回提供者引用

1. 从注册中心发现引用服务：

ReferenceConfig解析出来的URL格式为：

Registry://registry-host/com.alibaba.dubbo.registry.RegistryService?refer=URL.encode(“consumer://consumer-host/com.foo.FooService?version=1.0.0”)

基于扩展点的Adaptive机制，通过URL的”registry://”协议头识别，就会调用RegistryProtocol的refer方法，基于refer参数总的条件，查询提供者的URL，如：

Dubbo://service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过提供者URL的”dubbo://”协议头识别，就会调用DubboProtocol的refer()方法，得到提供者的引用

然后RegistryProtocol将过个提供者引用，通过Cluster扩展点，伪装成单个提供这引用返回

## 描述一个服务从发布到被消费的过程

### 3.1 服务暴露过程

|  |
| --- |
|  |

1. 将远程服务转化为invoker：首先ServiceConfig类拿到对外提供服务的实际类ref(HelloWorldImpl)，然后通过proxyFactory类的getInvoker方法使用ref生成一个AbstractProxyInvoker实例
2. Invoker转化为Exporter：
   1. Dubbo的实现：DubboProtocol的export()方法，它主要是打开socket侦听服务，并接收客服端发来的各种请求
   2. RMI的实现：RmiProtocol的export()方法，它通过Spring或Dubbo或JDK来实现RMI服务

### 3.2 服务消费过程

|  |
| --- |
|  |

1. 首先ReferenceConfig的init方法调用Protocol的refer方法生成invoker实例
2. Invoker转换为客户端需要的接口（HelloWorld）

## Dubbo超时机制和重连机制

超时机制（必须设置）的规则是如果在一定的时间内，provider没有返回，则认为本次调用失败；

重连机制：当调用接口失败后，我们可以设置重连的次数让程序继续重新连接接口，如果在指定的次数内都没有调通接口，则认为本次接口调用失败。

合理的配置超时时间：如果设置超时时间过短，一些复杂业务需要很长时间完成，导致在设定的超时时间内无法完成正常的业务处理，eg：发送邮件，当设置超时时间过短时，可能就会发出多分重复邮件，执行注册请求时，就会插入多条重复的数据

Dubbo指定接口和方法的超时配置：

|  |
| --- |
|  |

合理配置超时和重连次数：

1. 对于核心服务，可以去除重连机制，并重新设置超时时间
2. 业务处理代码必须放在服务端，客户端只做参数验证和服务调用，不涉及业务流程处理

## Dubbo服务请求失败如何处理

重连机制：dubbo在调用服务失败时，默认会重连两次，或者配置重连次数，如果在配置的调用次数内都失败，则认为此次请求失败，抛出异常

## 重连机制会不会造成错误

Dubbo的路由机制，会把超时的请求路由到其他机器上，而不是本机尝试，dubbo的重连机制也能一定程度上的保证服务的质量，但是必须合理配置重连次数。但是如果不合理的配置重试次数，当失败时会进行多次重试，这样在某个时间点出现性能问题，调用方连续重复调用，系统请求量变为正常值的n倍，系统压力增大，造成服务崩溃，所以需要根据业务情况规划好如何进行异常处理，何时进行重试及重试次数

## Dubbo负载均衡机制

Dubbo框架内置提供负载均衡的功能以及扩展接口，我们可以透明的扩展一个服务或服务集群，根据需要非常容易的增加或者移除节点，提高服务的可伸缩性，Dubbo框架内置提供了4种负载均衡策略

1. Random LoadBalance：随机策略，配置值为random，可以设置权重
2. RoundRobin LoadBalance：轮询策略，配置值为roundrobin
3. LeastActive LoadBalance：最不活跃优先策略，配置值为leastactive，根据请求调用次数，处理请求更慢的节点会受到更少的请求
4. ConsistenHash LoadBalance：一致性hash策略

## 分布式系统怎么做服务治理

Dubbo一般来说使用zookeeper注册中心来治理服务

SpringCloud使用Eureka注册中心来治理服务

Edas使用HSF注册中心来治理服务

## Zookeeper介绍、原理和使用场景

### 什么是zookeeper

Zookeeper是一个开源的分布式的应用程序的协调服务，它包含一个简单的原语集，分布式系统可以基于它实现同步服务、配置维护和命名服务等，它是集群的管理者，监视着集群中各个节点的状态，将简单易用的接口和性能高效、功能稳定的系统提供该用户

### zookeeper分布式与数据复制

Zookeeper作为一个集群提供一致的数据服务，他要在所有机器间做数据复制。

数据复制的好处：

1. 容错：一个节点出错，不至于让整个系统停止工作，别的节点可以接管它的工作
2. 提高系统的扩展能力：把负载分布到多个节点上，或者增加节点来提高系统的负载能力
3. 提高性能：让客户端本地访问就近的节点 ，提高访问速度

### Zookeeper角色描述

主要分为3类：领导者、学习者和客户端

|  |  |  |
| --- | --- | --- |
| 角色 | | 描述 |
| 领导者（Leader） | | 领导者负责投票的发起和决议，更新系统状态 |
| 学习者（Learner） | 跟随者（Follower） | Follower主要用来接收客户端请求并向客户端返回结果，在选主过程中参与投票 |
| 观察者（ObServer） | ObServer可以接收客户端连接，将写请求转发给leader节点，但ObServer不参与投票过程，只同步leader的状态。ObServer的目的是为了扩展系统，提高读取速度 |
| 客户端（client） | | 请求发起方 |

### Zookeeper服务与客户端

|  |
| --- |
|  |

### 9.6 Zookeeper设计目的

（1）最终一致性：client不论连接到哪个服务器，展示给他的都是同一个视图，这是zookeeper最重要的性能

（2）可靠性：具有简单、健壮、良好的性能，如果消息被一台服务器接受，那么它将被所有服务器接受

（3）实时性：zookeeper将保证客户端在一个时间间隔内获得服务器的更新信息，或者服务器失效信息。但由于网络延时等原因，zookeeper不能保证多个客户端能同时得到刚更新的数据，如果需要更新数据，应该在读数据之前调用sync()接口

（4）等待无关（wait-free）：慢的或者失效的client不得干预快速的client请求，使得每个client都能有效的等待

（5）原子性：更新只能成功或失败，没有中间状态

（6）顺序性：包括全局有序和偏序两种：全局有序是指在一台服务器上消息a在消息b发布前，则在所有server上消息a都将在消息b前发布；偏序是指如果一个消息b在消息a后被同一个发布者发布，a必将排在b前面

### 9.7 zookeeper下server工作状态

每个server有三个工作状态：

1. LOOKING：当前server不知道leader是谁，正在寻找
2. LEADING：当前server为选举出来的leader
3. FOLLOWING ：leader已经选出来，当前server与之同步

## Zookeeper选举流程

相关名词：

Serverid(sid)：在配置server时，给定的服务器的表示id

ZXID：服务器在运行时产生的数据id，zxid越大，表示数据越新

Epoch：选举的轮数

Server状态：LOOKING、LEADING、FOLLOWING、OBSERVING

### 服务器启动时间的Leader选举

若进行Leader选举，则至少需要两台机器，这里选取3台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器server1启动时，无法进行和完成Leader选举，当第二台服务器server2启动时，此时两台机器可以相互通信，每台服务器都试图找到Leader，于是进入Leader选举流程：

1. 每个server发出一个投票（启动期间：每个服务器的ZXID相同）：由于初始情况，server1和server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的sid和ZXID，使用(sid,ZXID)来表示，此时server1的投票为(1,0)，server2的投票为(2，0)，然后各自将这个投票发给集群中其他服务器
2. 接受来自各个服务器的投票：集群中的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票和是否是来自LOOKING状态的服务器
3. 处理投票：针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下：
   1. 优先检查ZXID，ZXID比较大的服务器优先作为Leader
   2. 如果ZXID相同，就比较sid，sid较大的服务器作为Leader

对于server1，它的投票是(1,0)，对于server2，它的投票是(2,0)，首先会比较两者的ZXID，均为0，再比较sid，此时server2的sid最大，于是更新自己的投票为(2，0)，然后重新投票，对于server2而言，其无需更新自己的投票，只是再次向集群中所有服务器发出上一次投票即可

1. 统计投票：每次投票后，服务器都会统计投票信息，判断是否有已经有过半机器接收到相同的投票信息，eg：(2,0)，对于server1和server2而言，都统计出集群中已经有两台服务器接收了(2,0)投票信息，此时便认为已经选出了Leader为server2
2. 改变服务器状态：一旦确定了Leader，每台服务器就要更新自己的状态，如果是Follower，那么久变更为FOLLOWING，如果是Leader，就变更为LEADING

### 服务器运行期间无法和Leader保持连接的选举

在zookeeper运行期间，Leader和非Leader服务器各司其职，即便当有非Leader服务器宕机或者加入，都不会影响Leader运行，但是一旦Leader服务器挂了，那么整个集群将暂停对外提供服务，进入新一轮的Leader选举，其过程和启动时期的Leader选举过程基本一致，若某一时刻Leader挂了，便开始Leader选举，过程如下：

1. 变更状态：Leader挂后，其余的所有服务器都会将自己的服务器状态变更为LOOKING，然后开始进入Leader选举过
2. 每个server发出投票（运行期间，各个服务器ZXID可能不同）：同启动期间
3. 接收来自各个服务器的投票：同启动期间
4. 处理投票：同启动期间
5. 统计投票：同启动期间
6. 改变服务器状态：同启动期间

## Zookeeper的用途，选举的原理是什么

### zookeeper用途

1. 命名服务：在zookeeper的文件系统里创建一个目录，就有唯一的path，通过path找到程序的部署机器
2. 配置管理：程序总是需要配置的，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。现在把这些配置全都放到zookeeper上，保存在zookeeper的某个目录节点上，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序都会收到zookeeper的通知，然后从zookeeper获取新的配置信息然后在应用的对应的系统中
3. 集群管理：是否有机器的退出和加入、选举master
   1. 机器的退出或加入：所有服务器约定在父目录GroupMembers下创建临时目录点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与zookeeper的连接断开，其所创建的临时目录点被删除，其他所有机器都受到通知
   2. 选举master：所有机器创建临时顺序编号目录节点，每次选取最小的机器作为master
4. 分布式锁
   1. 保持独占：我们将zookeeper上的一个znode作为一把锁，通过createZnode的方式来实现，所有客户端都去创建/distribute\_lock节点，最终成功创建的那个客户端也就拥有了这把锁。用完删除掉自己所创建的/distribute\_lock节点释放锁
   2. 控制程序：/distribute\_lock已经预先存在。所有客户端在他下面创建临时顺序编号目录节点，和master一样，编号最小的获取锁，用完删除
5. 队列管理
   1. 同步队列：当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有的成员到达
   2. 队列按照FIFO的方式进行入队和出队的操作

### Zookeeper选举原理

Zookeeper的核心是原子广播，这个机制保证了各个server之间的同步，实现这个机制的协议叫做zab协议。Zab协议有两种模式：恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，zab就进入了恢复状态，当领导着备选出来后，且大多数server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了server和leader有相同的系统状态

为了保证事务的顺序一致性，zookeeper采取了递增的事务id号（zxid）来标识事务，所有的协议（protocol）都在被提出的时候加上zxid，zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位属于递增计数

## Zookeeper的watch(通知)机制

在zookeeper中引入了watcher机制来实现分布式的通知功能，客户端注册监听它所关心的目录节点，当目录节点发生变化（数据发生变化，子目录节点被增加或者删除），zookeeper会通知客户端。zookeeper允许客户端向服务端注册一个watcher监听，当服务器的一些特定事件触发了这个watcher，那么就会向指定的客户端发送一个事件通知来实现分布式的通知功能。客户端对zookeeper的某个事件上添加Watcher时，客户端会保持一个EventThread线程接收事件通知，根据绑定的event来触发watcher执行process方法，watcher是一次性的，只会触发一次

Zookeeper的watcher机制主要包括三部分：

1. zookeeper：部署在远程服务器上的zookeeper单机或集群
2. Client：客户端线程
3. WatchManager：接口，用于管理各个监听器。

客户端在向zookeeper服务器注册Watcher的同时，会将watcher对象存储在客户端的WatcherManager中，当zookeeper服务器触发Watcher事件后，会向客户端发送通知，客户端线程从WatcherManager的实现类中取出对应的Watcher对象来执行回调逻辑

## MQ分类，优缺点

### RabbitMQ

RabbitMQ是基于Erlang语言编写的开源消息队列，通过Erlang的Actor模型实现了数据的可靠稳定传输。

RabbitMQ遵循AMQP（高级消息队列协议）协议，RabbitMQ的broker由Exchange、Binding、queue组成，其中Exchange和Binding组成了消息的路由键

Producter通过连接channel和server进行通信，consumer从queue获取消息进行消费（长连接，queue有消息会推送到consumer，consumer循环从输入流获取数据）。

RabbitMQ以broker为核心，有消息确认机制

因为其可扩展性，可以通过插件的形式使用STOMP、XMPP、AMQP1.0、SMTP，还可以通过插件使用HTTP这种非消息的传输协议

因为其实现了代理（Broker）架构，意味着消息在发送到客户端之前可以在中央节点上排队

此特性使得RabbitMQ易于应用和部署，适用于路由、负载均衡和消息持久化等场景，用消息队列只需几行代码即可搞定。但是，这使得它扩展性很差、速度较慢，因为中央节点增加了延迟，消息封装后也比较大，如果要配置RabbitMQ则需要安装Erlang环境

RabbitMQ支持miror的queue，主queue失效，miror queue接管。

RabbitMQ的负载均衡需要单独的load balance来实现。

RabbitMQ在吞吐量方面不如Kafka，RabbitMQ支持对消息的可靠传输，支持事务，不支持批量操作，基于存储的可靠性的要求存储可以采用内存或者硬盘

总的来说，RabbitMQ在数据一致性、稳定性和可靠性方面比较优秀，而且直接或间接的支持多种协议，对多种语言支持良好，但是其性能和吞吐量不好，由于Erlang语言本身的限制，二次开发成本较高。

### Kafka

Kafka是Linkedin于2010年12月开发并开源的一个分布式流平台，现在是Apache的顶级项目，是一个高性能跨语言分布式发布/订阅（Publish/Sbuscribe）消息队列系统，以pull的形式消费消息

具有以下特性：消息持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上就可以达到10W/S的吞吐速率；完全的分布式系统，Broker、Producter、Consumer都原生自动支持分布式，自动实现负载均衡；因为设计之初是作为日志流平台和运营消息管道平台，所以实现了消息顺序和海量堆积

Kafka遵从一般的MQ结构，producter、broker、consumer，以consumer为中心，消息的消费信息保存在客户端consumer上，consumer根据消费的点，从broker上批量拉取（pull）数据；无消息确认机制。

Kafka具有高吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序操作，具有O(1)的复杂度，消息处理的效率很高

Kafka的broker支持主备模式

Kafka采用zookeeper对集群中的broker、consumer进行管理，可以注册topic到zookeeper上，通过zookeeper的协调机制，producter保存对应topic的broker信息，可以随机或者轮询发送到broker上；并且producter可以根据语义指定分片，消息发送到broker的某分片上

支持hadoop数据并行加载，对于像hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案，kafka通过hadoop的并行加载机制来统一了在线和离线的消息处理

Kafka相对于ActiveMQ是一个轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统

Kafka自身服务与消息的生产和消费都依赖于zookeeper，使用scata语言开发，因为其消息的消费使用客户端pull方式，消息可以被多个客户端消费，理论上消息会重复，但是不会丢失（除非消息过期），常用于作为日志传输的消息平台的场景

### ZeroMQ

ZeroMQ号称是“史上最快的消息队列”，尤其针对大吞吐量的需求场景，基于c语言开发的，可以在任何平台上通过任何代码连接，通过inproc、IPC、TCP、TIPC、多播传送消息，支持发布/订阅、推---拉、共享队列等模式，高速异步I/O引擎。

ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色，你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后就可以实现程序之间的消息传送了

但是ZeroMQ只支持非持久性的队列，如果down机，数据将丢失。

根据官方说法，ZeroMQ是一个简单好用的传输层，像框架一样的可嵌入的socket类库，使socket编程更加简单、简洁、性能更高，是专门为高吞吐量、低延迟的场景开发的，ZeroMQ与其它MQ有着本质的区别，它根本不是消息队列服务器，更类似与一个底层网络通讯库，对原有Socket API进行封装，在使用的时候引入对应的jar包即可。同时，如果我们想要作为消息队列使用的话，需要开发大量代码，而且，ZeroMQ不支持消息持久化，并不是安全可靠的消息传输，所以需要自己编写代码保证消息的可靠传输

### ActiveMQ

ActiveMQ是Apache下的一个子项目，类似于ZeroMQ，它能够以代理人和点对点的技术实现队列，可以部署于代理模式和P2P模式，同时类似于RabbitMQ，少量的代码就可以高效的实现高级应用场景而且只需付出低消耗

支持OpenWrie、Stomp、AMQP v1.0、MQTT v3.1、REST、Ajax、webservice等多种协议；完全支持JMS1.1和J2EE1.4规范（事务、持久化）；支持持久化到数据库。重量级，对于队列较多的情况支持不好，有可能丢失消息

### RocketMQ（ONS）

RocketMQ是ali开源的消息中间件，使用纯JAVA开发，具有高吞吐量、高可用性、适用于大规模分布式系统应用的特点，RocketMQ思路起源于Kafka，它对消息的可靠传输以及事务做了优化，目前在ali被广泛用于交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景，支撑了ali多次双十一活动。

因为是ali内部从实践到产品的产物，因此里面很多接口、API并不是很普遍使用，比kafka性能更强、支持海量堆积

## MQ的连接是线程安全吗

不安全

## 数据的垂直拆分和水平拆分

### 垂直拆分（专库专用）

垂直拆分指的是按照业务分类将不同模块的表分布到不同的数据库上面（后台管理、用户模块、订单模块等）

优点：（1）拆分后业务清晰，业务规则明确

（2）系统之间容易整合和扩展

（3）数据维护简单

（4）解决了表与表之间的IO竞争

缺点：（1）部分业务表无法使用join查询，只能通过接口方式获取，提高了系统复杂度

（2）存在单表数据瓶颈

### 水平拆分

水平拆分指的是将同一个表拆分到不同的表或数据库中（按照某个字段的某种规则，如用户表的性别字段，男女各一张表）

优点：（1）不存在单表大数据量

（2）按照合理拆分规则拆分，join操作避免垮裤

缺点：（1）数据多次扩展难度和维护量大

（2）拆分规则难以抽象

（3）表与表之间存在IO竞争

## Redis/zk节点宕机如何处理

做备份：每个节点都有备份界点，当一个节点和其所有备份节点都挂的时候，集群也就停止提供服务了

## 分布式集群下如何做到唯一序列号

1. 数据库自增字段ID（如果表做了水平分表就不能使用了，此时ID会重复）
2. 专门新建一张表，利用序列或者自增专门存储唯一序列号
3. 利用UUID或者时间戳等

## 分布式锁的实现

分布式锁：一种跨服务器（JVM）控制共享资源访问的互斥机制。在分布式系统环境下，一个方法在同一时间只能被一台机器的一个线程执行

（1）基于缓存（Redis）实现分布式锁：获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁。

eg：获取名为foo的锁：

SETNX lock.foo <current Unix time + lock timeout + 1>

如果返回1，说明获取成功

如果返回0，说明该锁被其他线程获取，可以进行重试

（2）基于数据库实现分布式锁：在数据库中创建一张表，表中包含方法名字段，并在方法名字段上创建唯一索引，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

（3）基于zookeeper实现分布式锁：

a.创建一个目录dislock

b.线程A想要获取锁就在dislock目录下创建临时顺序节点

c.获取dislock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁

d.线程B获取所有节点，判断自己是不是最小节点，设置监听比自己次小的节点

e.线程A处理完，删除自己的节点，线程B监听到变更时间，判断自己是不是最小的节点，如果是，则获得锁

## 接口的幂等性的概念

背景：系统大多拆分为分布式SOA，或者微服务，一个系统中包含了多个子系统服务，而一个子系统服务往往回去调用另一个服务，而服务调用服务无非就是使用RPC通信或者Restful，既然是通信，那么就有可能在服务器处理完毕后返回结果的时候挂掉，这个时候客户端发现很久没有反应，那么会多次点击按钮，这样就会请求多次，在请求多次的时候需要保证处理数据的结果同意，所以就幂等性

幂等性：就是用户对于同一操作发起一次请求或者多次请求的结果是一致的，不会因为多次请求而产生了错误的结果。eg：支付场景，用户支付完后，出现网络异常，此时钱已经扣了，用户再次点击按钮，就需要保证第二次付款失败，这就用到了接口的幂等性

设计接口做到幂等（支付场景）：

1. 当多次点击支付按钮进行付款时，首先查询订单支付状态
2. 如果已经支付，直接返回结果
3. 如果未支付，则支付扣款并且保存流水
4. 返回支付结果

## 全局ID

如果要使用全局唯一ID，就是根据业务的操作和内容生成一个全局ID（这个生成的全局唯一ID需要保证不重复，eg：用户编号+随机数等等），在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。如果不存在，说明该方法未执行，则把这个全局ID存储到数据库中，如果存在表示该方法已经执行

## 分布式项目中如何做事务管理

分布式事务特性CAP

C(Consistency)：一致性。客户端知道一系列的操作都会同时执行A(Availability)：可用性。每个操作都必须以可预期的响应结束

P(Partition tolerance)：分区容错性。即使出现单个组件无法可用，操作依然可以完成

分布式事务特性BASE

BA(Basically Available)：基本可用。

S(Soft state)：软状态。

E(Eventually consistant)：最终一致性。

### 21.1 基于XA协议的两阶段提交方案(2pc)

实现方式：使用两阶段来完成一个全局事务提交，XA规范的基础是两阶段提交协议。第一阶段是表决阶段，所有参与者都将本事务能否成功的信息反馈发给协调者；第二阶段是执行阶段，协调者根据所有参与者的反馈，通知所有参与者，步调一致的在所有分支上进行提交或回滚

优点：尽量保证了数据的强一致，适合对数据强一致要求很高的领域

缺点：实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景

### 21.2 TCC(补偿事务)

使用方式：TCC方式是两阶段提交的一种改进。其将整个业务逻辑的每个分支显示的分成了Try（完成事务的准备工作）、Confirm（完成事务的提交）、Cannal（完成事务的回滚）三个操作

优点：跟2PC相比，实现以及流程相对简单，但数据一致性不如2PC

缺点：TCC属于应用中的一种补偿方式，需要在实现的时候多写补偿的代码

### RocketMQ(消息队列)

实现方式：将本地操作和发送消息放到一个事务中，保证本地操作和消息发送要么两者都成功要么都失败。下游应用向消息系统订阅该消息，接收到消息后执行相应操作

缺点：代码侵入性高，业务复杂

### GTS

GTS是由ali提供的一款分布式事务中间件，可以为微服务架构中的分布式事务提供解决方案，包括dubbo、edas、springCloud

实现方式：GTS包括客户端（GTS Client）、资源管理器（GTS RM）和事务协调器（GTS Server）三个部分。client主要用来定义事务边界，完成事务的发起与结束。rm完成事务分支的创建、提交、回滚等操作。Server主要负责分布式事务的整体推进，事务生命周期的管理。使用注解@TxcTransaction可开启分布式事务

优点：

（1）性能强大：解决了事务ACID特性与高性能、高可用、低侵入等问题，单事务分支的平均响应时间在2ms左右

1. 侵入性低：使用注解@TxcTransaction即可开启分布式事务，业务与事务分离，将微服务从事务中解放出来，微服务关注业务本身，不在需要考虑反向接口、幂等、回滚等问题，降低了开发的难度及工作量
2. 当调用的第三方接口没有接入GTS时，可以使用GTS的MT模式(等价于TCC模式)，可以根据业务需求定义每个事务的具体行为
3. 解决了XA事务协调器的单点问题，可以保证各种异常情况下的数据严格一致

# 数据库

## oracle数据库

### 2.1 存储过程

创建存储过程：

create or replace procedure 存储过程名

|  |
| --- |
|  |

程序调用：

Mybatis调用存储过程

|  |
| --- |
|  |

传入参数使用map，然后会将查询结果返回到map里面

|  |
| --- |
|  |

### 2.2 触发器

用来记录表数据日志，将对表的CRUD操作记录到日志表中

create or replace trigger 日志名；

### 2.3 Databases links

通过Databases links 可以连接到其他服务下的数据库表

### 2.4 oracle中sql

（1）递归查询(树结构)：查询一个菜单下面所有子菜单

使用start with ... connect by prior语法

select \* from tableName start with 条件一connect by prior 条件二

|  |
| --- |
|  |

其中start with可以指定一个或多个根节点

1. 分页

page：页码，size：每页显示条数

select \*

from (select t.\*,rownum rn

from table\_name

order by id desc) t

where rownum <= page\*size)

where rn > (page-1)\*size

1. 行转列，列转行

pivot和unpivot的用法

## mysql数据库

### Mysql性能优化

#### 数据查询优化

1. 使用EXPLAIN查询sql执行情况：EXPLAIN select \* from table

|  |
| --- |
|  |

Select\_type：select的查询类型，常用取值有：simple最常见

Simple：表示此查询不包含union查询或子查询

Primary：表示此查询是最外层的查询

Union：表示此查询是union的或者随后的查询

Dependent union：union中的第二个或后面的查询语句，取决于外面的查询

Union resule：union的结果

Subquery：子查询中的第一个select

Dependent subquery：子查询中的第一个select，取决于外面的查询，即子查询依赖于外层的查询结果

Table：查询的表

表示查询涉及的表或衍生表

Partitions：匹配的分区

表示是否使用了分区，null表示没有分区

Type：判断查询是否高效的重要依据，type字段可以判断是全表扫面还是索引扫描。

System：表中只有一条数据，这个类型是特殊的const类型

Const：针对主键或唯一索引的等值查询扫描，最多只返回一行数据。Const查询速度非常快，因为它仅仅读取一次、

Eq\_ref：此类型通常出现在多表的join查询，表示对于前表的每一个结果，都只能匹配到后表的一行结果，并且查询的比较操作通常是=，查询效率较高

Ref：此类型通常出现在多表的join类型，针对于非唯一或非主键索引，或者是使用了最左前缀规则索引的查询

Range：表示使用索引范围查询，通过索引字段范围获取表中部分数据记录，这个类型通常出现在=、<>、>、>=、<、<=、is null、between、in操作中。当type是range时，那么explain输出的ref字段为null，并且key\_len字段是此次查询中是用到的索引的最长的那个

Index：表示全索引扫描（full index scan）和all类型类似，只不过all类型是全表扫描，而index类型则仅仅扫描所有的索引，而不扫描数据。Index类型通常出现在：所要查询的数据直接在索引树种就可以获取到，而不需要扫描数据，但是这种情况时，extra字段会显示using index

All：表示全表扫描，这个类型的扫描是性能最差的查询之一。通常来说，我们的查询不应该出现all类型的查询，因为这样的查询在数据量大的情况下，对数据库的性能是巨大的灾难。如一个查询是all类型的查询，那么加索引来避免

性能：all<index<range<ref<eq\_ref<const<system

Possible\_keys：此次查询中可能选用的索引

Key：此次查询中确切用到的索引

Ref：哪个字段或常数与key一起被使用

Rows：显示此查询一共扫描了多少行，这个是一个估计值

Filtered：表示此查询条件所过滤的数据的百分比

Extra：额外的信息

1. 使用索引

注意：有些情况会使索引失效，使用EXPLAIN查看sql执行情况时，possible\_keys字段可以查看索引是否起作用

1. 使用like模糊查询的时候：如果%为匹配字符串的第一个字符，索引不起作用，只有%不在第一个字符的时候，索引才生效
2. 使用联合索引查询的时候：mysql可以为多个字段创建索引，一个索引可以包含16个字段。对于联合索引，只有查询条件中使用了这些字段中的第一个字段时，索引才会生效
3. 使用or关键字查询的时候：查询语句的查询条件中只有or关键字，且or前后的两个条件的列都是索引时，索引才会生效
4. 子查询优化

执行子查询时，mysql需要创建临时表，查询完毕后再删除这些临时表，所以子查询效率会受到影响

优化：可以使用join来代替子查询

#### 数据插入优化

插入数据时，影响插入速度的主要原因是索引、唯一性校验及插入条数等。使用不同的存储引擎对数据插入的优化方式不同

1. Myisam
2. 禁用索引

对于非空表：插入记录时，mysql会根据表的索引对插入的记录建立索引。如果插入大量数据，建立索引就会降低数据插入速度

插入数据时禁用索引，数据插入成功后再启用索引

禁用索引：alter table tableName disable keys

启用索引：alter table tableName enable keys

对于空表批量插入数据，则不需要进行此操作，因为Myisam引擎的表是在导入数据后才建立索引

1. 禁用唯一性校验

唯一性校验会降低数据插入速度，可以在插入数据之前禁用唯一性校验，数据插入完成后再开启

禁用唯一性校验：set unique\_checks = 0;

启用唯一性校验：set unique\_checks = 1;

1. 插入数据时，可以使用一个insert语句代替多个insert语句

Insert into tableName() values(),(),()......

Insert into tableName() select \* from tableName

1. Innodb
2. 禁用唯一性校验：同Myisam的禁用唯一性校验
3. 禁用外键校验

数据插入之前可以禁用外键校验，数据插入完成后再启用，可以提高插入速度

禁用外键校验：set foreign\_key\_checks = 0;

启用外键校验：set foreign\_key\_checks = 1;

1. 禁止自动提交

插入数据之前禁止事务的自动提交，数据插入完成后再恢复，可以提高插入速度

禁止自动提交：set autosubmit = 0;

启用自动提交：set autosubmit = 1;

#### 数据库结构优化

一个好的数据库设计方案对于数据库的性能会起到事半功倍的效果，需要考虑数据冗余、查询和更新速度、字段数据类型是否合理等

1. 将字段很多的表分解成多个表

对于字段很多的表，如果有些字段的使用频率很低的话，可以将这些字段分离出来形成一张新表。因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

1. 增加中间表/视图

如果需要经常联合查询的表，可以建立中间表/视图提高查询效率

通过建立中间表/视图，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

#### 服务器优化

1. 优化服务器硬件
2. 优化mysql的参数

mysql的配置参数都在my.conf或者my.ini文件的[mysqld]组中。eg：

### Mysql数据库存储引擎

#### Innodb

1. 存储结构：Innodb所有的表都保存在同一个文件中（也可能是多个文件，或者是独立的表空间文件）。Innodb表的大小之受限于操作系统文件的大小，一般为2GB
2. 存储空间：Innodb需要更多的内存和存储空间。它会在主内存中建立其专用的缓存池用于高速缓存数据和索引
3. 事务支持：支持事务。具有事务提交（commit）、回滚（rollback）和崩溃修复能力（crash recovery capabilities）的事务安全型表
4. 外键：支持外键
5. CRUD操作：如果执行大量的insert、update或者delete，处于性能方面的考虑，应该选择Innodb（支持行级锁，增删改的时候只需要锁定改变的行就行）。但delete from table时，Innodb不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table 这个命令

#### Myisam：默认使用

1. 存储结构：每个Myisam在磁盘上被存储成3个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD（MYData）。索引文件的扩展名为.MYI（MYIndex）
2. 存储空间：Myisam可以被压缩，存储空间较小。支持3种不同的存储格式：静态表（默认，但是注意数据末尾不能有空格，会被去掉）、动态表、压缩表
3. 事务支持：不提供事务支持。强调的是性能，每次查询具有原子性，其执行速度比Innodb类型更快
4. 外键：不支持外键
5. CRUD操作：如果执行大量的select，Myisam是最好的选择（因为没有支持行级锁，只支持表级锁），在增删改的时候需要锁定整个表格，效率会低一些

#### Innodb和Myisam的选择

因为Myisam相对简单，所以在效率上优于Innodb，如果系统读多写少，优先选择Myisam，而且Myisam恢复速度快。

如果系统读少写多的时候，尤其是高并发的时候，优先选择Innodb

#### Innode和Myisam的使用

|  |
| --- |
|  |
|  |

### mysql中sql

1. 分页

pageNumber：页码，pageSize：每页显示条数

select \* from table\_name limit 9,4 从第10条开始，返回4条数据

select \* from table\_name limit(pageNumber-1)\*pageSize,pageSize

1. 对表中某个字段分组后排序取第一条

Select \* from table where id in(

Select max(id) from table group by column

)

eg:对rule\_id分组后取最新的一条（create\_time最大或者id最大均为最新的一条）

|  |
| --- |
|  |

## 数据库分库分表策略，分库分表后，如何解决全表查询问题

### 分表

当数据比较大的时候，对数据进行分表操作。分表就是将一张表的数据平均分配到多张表。表容量：表的数量

eg：将user表分为user\_1、user\_2 ...等多张表中

假设这里有5张表也就是说表容量为5，则我们在进行存储数据的时候，首先对用户id进行取模操作，如果userId为UUID，则先通过UUID进行hash获取到UUID的整数值，根据userId%5+1获取对应的表进行存储查询操作。

假如userId%5+1=2，则在查询的时候：

Select \* from user\_2 where userId = 6

其中的2作为动态参数加入：mybatis中

|  |
| --- |
|  |

### 分库

根据不同的业务功能将表存放到不同的数据库

## 事务特性和隔离级别

### 事务特性:ACID

### 隔离级别

1. Read Uncommited（未提交读）：也称作脏读，事务还没提交，其它事务也可以看到

eg：售票系统里面只有一张票，A接到订单要售出一张票，此时查看售票系统里面有一张票，于是要售出的时候，也就是说第一不执行完毕（查看售票系统里面有一张票）第二步正在执行的时候（售票过程中），B也接到订单，也看到还有一张票，B也售出，此时就出现1张票出售两次的情况。而数据库为了保证事务，所以一张票都没有出售

1. Read Commited（提交读）：一个事务的开始，只能看到已经完成的事务的结果，正在执行的事务，无法被其他事务看到。大多数数据库的隔离级别默认使用Read Commited。这种隔离级别会造成读取到旧数据的问题

eg：售票系统里面只有一张票，A接到订单要售出一张票，此时查看售票系统里面有一张票，于是要售出的时候，也就是说第一步执行完毕（查看售票系统里面有一张票）第二步正在执行的时候（售票过程中），B也接到订单，也看到还有一张票，此时B看不到A正在执行的事务

1. Repeatable Read（可重复读）：解决了数据脏读的问题，该级别保证了每行的数据的结果是一致的，也就是上面说到的旧数据。但是这种隔离级别解决不了幻读（就是突然新增的数据。就是某个事务在读取某个表的数据，但是另一个事务又向这个表插入数据，导致多次读取数据的行数不一致）的问题。
2. Serializable（可串行化）：最高的隔离级别，通过强制事务串行执行，避免了前面幻读的问题，由于加锁，导致大量的请求超时，因此性能比较低下。并发量低并且对数据一致性要求高的时候用。

查询当前数据库的事务隔离级别：

|  |
| --- |
|  |

设置数据库隔离级别：

|  |
| --- |
|  |

# JVM（Java虚拟机）内存模型

一般将内存分为堆内存和栈内存

## 栈内存

栈是与每一个线程相关联的，JVM在创建每一个线程的时候，会分配一定的栈空间给线程，它主要用来存储线程执行过程中的局部变量、方法返回值以及方法调用上下文，栈内存随着线程的终止而释放

## 堆内存

堆内存是所有线程共享的一块最大的内存区域，也是垃圾收集器管理的主要区域，用来保存各种Java对象，比如数组、线程对象等

|  |  |  |
| --- | --- | --- |
|  | 堆内存 | 栈内存 |
| 存放的数据类型 | 通过new关键字创建的对象或数组（在堆中分配内存）、包装类、每个变量都有默认值，所以final修饰的变量存在堆中 | 基本类型的变量和对象的引用变量、局部变量、方法返回值和方法调用上下文 |
| 存取速度 | 运行时动态分配内存，存取速度慢 | 存取速度快，仅次于CPU |
| 生命周期 | 动态分配内存大小，生命周期不必事先告诉编译器 | 栈中的数据大小和生命周期必须是确定的，缺乏灵活性 |
| 可见性 | 线程共享的内存区域 | 线程私有数据区域 |
| 垃圾回收 | 使用完毕后并不会立即进行垃圾回收，会在垃圾回收器空闲时回收 |  |

## 堆的分区

Java堆一般又可以分为3部分：年轻区、年老区、永久区



1. Young：年轻区（代）

Young又被分为3部分：Eden区（对象优先分配）和两个大小相同的Survivor区，其中Survivor区某一时刻只有一块被使用，另一块留作垃圾回收时复制对象用，在Eden区变满的时候，GC就会将存活的对象转移到Survivor区间中，根据JVM策略，在经过几次垃圾收集后，仍然存活于Survivor区的对象将被移动到Tenured区中

1. Tenured：年老区

Tenured区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在Young区中复制转移一定的次数后，对象就会被转移到Tenured区中，如果系统中用了application级别的缓存，缓存中的对象往往会被转移到Tenured区中，垃圾回收频率较低，而且回收的速度也比较慢

1. Perm：永久区(注意：jdk1.8之后，将最初的永久代取消了， 由元空间取代)

Perm区主要保存虚拟机加载的类信息(class)、常量池、静态变量、即时编译后的代码、method、filed对象等，这部分的空间一般不会内存溢出，除非一次性加载了很多的类，不过在涉及到热部署应用的服务器的时候，有时候会遇到java.lang.OutOfMemoryError : PermGen space(metaSpace)的错误，造成这个错误的很大原因有可能是每次都重新部署后，但是重新部署后，有些旧的class没有被卸载掉，这样就造成了大量的class对象被保存到Perm区中，这种情况下，一般重新启动应用服务器可以解决问题

1. Virtual区：

最大内存和初始内存的差值就是Virtual区

## 设置堆内存中各个区大小

可以在tomcat的catalina.sh中添加JAVA\_OPTS进行设置，见tomcat内存优化

## 内存泄露、内存溢出

### 内存泄露和内存溢出区别

内存泄露（memory leak）：

内存泄露指在程序申请内存后，无法释放已申请的内存空间，一次内存泄露可以忽略，但发生多次内存泄露后，最终会消耗尽系统所有内存，内存泄露会导致内存溢出

内存溢出（out of memory）：

内存溢出指在程序申请内存时，没有足够的内存空间供其使用，就是你要求分配的内存超过了系统能给你的，系统不能满足要求，产生内存溢出，eg：申请了一个Integer，但你需要一个Long存储数据

### 常见的内存溢出

1. 栈溢出：java.lang.stackoverflowerror
2. 堆溢出：java.lang.OutOfMemoryError:java heap space
3. Java.lang.OutOfMemoryError:GC overhead limit exceeded

GC回收时间过长：超过98%的时间用来做GC并且回收了不到2%的堆内存

1. Java.lang.OutOfMemoryError: Direct buffer memory

分配到本地内存，本地内存不足，导致程序崩溃

1. Java.lang.OutOfMemoryError:unable to create new native thread

创建的线程数量超过了系统默认允许的最大线程数量

Linux中非root用户默认1024个线程，root用户无上限

修改配置文件：vim /etc/security/limits.d/90-nproc.conf

添加一行配置，用户名及对应的最大线程数

1. Java.lang.OutOfMemoryError:Metaspace

### 内存溢出的原因及解决办法

原因：

1. 启动参数内存值设定过小
2. 代码中存在死循环或循环产生过多重复的对象实体
3. 内存中加载的数据量过于庞大，如一次性从数据库中取出所有数据
4. 集合类中有对对象的引用，使用完后未清空，使得JVM不能回收

解决方案：

1. 修改JVM的启动参数，直接增加内存
2. 查看日志，看内存溢出（out of memory）前是否有其它异常
3. 代码走查和分析，找出可能发生内存溢出的位置
4. 检查对数据库的查询，是否有一次获得全部数据的查询，一般来说，如果一次取过多的记录（十万条）到内存，就可能引起内存溢出，这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据较多，一次查询就可能造成内存溢出。因此对于数据库查询采用分页查询方式。
5. 检查代码中是否有死循环或者递归
6. 检查代码中是否有大循环重复产生新对象实体
7. 检查list、map等集合对象是否有使用完后，未清除的问题。List、map对象会始终存有对对象的使用，使得这些对象不能被GC回收
8. 使用内存查看工具动态查看内存使用情况

### 内存泄露产生原因及解决方案

产生原因：

1. 建立连接后没有关闭

eg：try--catch--finally 中在try中建立连接，在finally及时释放连接

1. 单例模式：如果单例对象持有外部的引用，那么这个对象将不能被JVM正常回收，导致内存泄露
2. 静态集合类引起的内存泄露

|  |
| --- |
|  |

此时对象o被vector使用，所以o=null对GC来说是不可回收的，应该将vector设为null

## JVM调优

### JVM的参数类型

#### 标配参数

1. java -version
2. java -help
3. java -showversion

#### X参数

（1）-Xint：解释执行

|  |
| --- |
|  |

（2）-Xcomp：第一次使用就编译成本地代码

|  |
| --- |
|  |

（3）-Xmixed：混合模式

#### XX参数

1. 布尔类型

-XX:+或者-某个属性 +表示开启true -表示关闭false

eg：使用jps和jinfo命令查看某个正在执行的程序的jvm的参数是否开启？具体参数值是多少？

jps -l：查看进程编号

jinfo -flag 配置项 进程编号：查看对应配置项的参数配置

(或者：jinfo -flags 进程编号：查看该进程下所有的参数配置)

查看是否开启GC收集细节：

-XX:-PringGCDetails -XX:+PringGCDetails

|  |
| --- |
|  |

查看是否使用串行垃圾回收器 :

-XX:-UseSerialGC -XX:+UseSerialGC

1. kv键值对类型：-XX:属性key=属性值value

|  |
| --- |
|  |

-Xms 1024m 等价于-XX:InitialHeapSize 初始化堆内存大小

-Xmx 1024m 等价于-XX:MaxHeapSize 最大堆内存大小

## 6.2 查看JVM参数值

（1）jps -l 和 jinfo -flag

（2）查看jvm默认初始化值：java -XX:+PrintFlagsInitial

|  |
| --- |
|  |

1. 查看被修改过后的值：java -XX:+PrintFlagsFinal

=表示初始化值，没有被修改过的

:=表示被修改过后的值，可能是jvm修改过或者人为修改

1. 查看值的过程中并修改某个值：

java -XX:+PrintFlagsFinal -XX:Xms1024m T

1. java -XX:+PrintCommandLineFlags -version：可以更方便的看使用了哪种垃圾回收器(最后一个参数)

## JVM常用基本参数配置

（1）-Xms1024m 等价于-XX:InitialHeapSize

初始化内存大小，默认为物理内存的1/64

（2）-Xmx1024m 等价于-XX:MaxHeapSize

最大分配内存，默认为物理内存的1/4

（3）-Xss1024k 等价于-XX:ThreadStackSize

设置单个线程栈空间的大小，跟平台有关，linux一般默认为512k~1024k，windows默认为0

（4）-Xmn1024m 等价于

设置年轻代大小

（5）-XX:MetaspaceSize：设置元空间大小

元空间的本质和永久代类似(jdk1.8前有永久代，jdk1.8将永久代用元空间替代)，都是对JVM规范中方法区的实现。不过元空间与永久代最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间大小仅受本地内存大小限制

（6）-XX:SurvivoRatio：设置新生代中eden和s0/s1区空间的比例

SurvivoRatio值就是设置eden区的比例占多少，s0和s1区相同

默认：-XX:SurvivoRatio=8 即eden:s0:s1 = 8:1:1

设置：-XX:SurvivoRatio=4 即eden:s0:s1 = 4:1:1

（7）-XX:NewRatio：配置年轻代与老年代在堆结构的占比

NewRatio值就是设置老年代的占比，剩下的1给新生代

默认：-XX:NewRatio=2，新生代(1/3)与老年代(2/3)比例为1:2

设置：-XX:NewRatio=4，新生代(1/5)与老年代(4/5)比例为1:4

（8）-XX:MaxTenuringThreshold：设置垃圾最大年龄，即从年轻代进入年老代要经过几次垃圾回收，默认15次

默认：-XX:MaxTenuringThreshold=15

Jdk1.8后取值只能[0,15]之间，如果设置为0，则直接进入老年代

（9）-XX:+PrinrGCDetails：打印详细GC收集日志信息

|  |
| --- |
|  |

GC日志信息解读(分析)

|  |
| --- |
|  |

# JMM（Java内存模型）

Java内存模型（Java Memory Model，简称JMM）是一种抽象的概念本身并不存在，它描述的是一组规则或规范，通过这组规则定义了程序中各个变量（包括实例字段、静态字段和构成数组对象的元素）的访问方式

|  |
| --- |
|  |

## 方法区（Method Area）

线程共享内存区域，主要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，通常和永久代（perm）关联在一起，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。方法区中存在一个叫运行时常量池的区域，它主要用于存放编译器生成的各种字面量和符号引用，这些内容将在类加载后存放到运行时常量池中，以便后续使用

## 堆（Heap）

线程共享的内存区域，在虚拟机启动时创建，主要用于存放对象实例，几乎所有的对象实例都在这里分配内存，是垃圾收集器的主要管理区域，因此很多时候也被称为GC堆，如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常

## 虚拟机栈（JVM Stacks）

线程私有数据区域，生命周期和线程相同，每一个方法执行时都会创建一个栈帧来存储方法的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。每个方法从调用到结束就是一个栈帧在虚拟机栈中的入栈和出栈过程

## 程序计数器（Program Counter Register）

线程私有区域，内存空间小，是当前线程所执行的字节码的行号指示器，每个线程都要有一个独立的线程计数器，线程的阻塞、恢复、挂起等一系列操作都需要程序计数器的参与

## 本地方法栈（Native Method Stacks）

线程私有区域，主要为虚拟机使用到的Native方法服务

# 面试

## 1.自我介绍

您好，我叫刘润恺，我今年28岁，来自于山西省平遥县，于2012年毕业于山西工商学院。在这几年的工作经验当中，通过自己的不断努力，使得自己学习到了非常多的技能，并且拥有良好的编程风格。我个人性格沉稳而且为人谦和，具有较强的适应能力，所以与公司同事可以融洽相处，并且具有较强的学习能力以及多年的工作经验，所以我相信我能够胜任贵公司的工作。

最后，希望得到您的（面试官的）点播，预祝此次面试顺利。

## 离职原因

从保险的角度看，一般从职业发展、技能提升的角度来回答是最好的，可以体现一下对公司的不舍

我在这家单位工作了2年多了，虽然员工之间相处融洽，公司管理方面也挺好的，但是工作的内容和职务没有什么变化，短期内公司的也不会有大的变化，为了个人的职业生涯着想以及提升自己的技能，换一家单位（如果面试区块链的话，可以说是为了换个发展方向，因为区块链在最近的几年内发展非常迅速）

## 期望薪资

我的期望薪资是15k，我相信贵公司一定会给我一个合理的薪资

## 学校情况

12年毕业于山西工商学院，二本，当年分数线440，考了452

学校地址：太原市小店区邬城南路

校训：诚信、奉献、拼搏、争先

学校所设专业：计算机科学与技术、传媒、建筑、金融等专业

4岁上大班

小学：1997-2002 5岁上小学

初中：2002-2005

高中：2005-2008

大学：2008-2012