# java 虚拟机、内存、加载器

## 1.1java的跨平台原理

java跨平台是通过JVM（java 虚拟机）实现的

JVM（java虚拟机）不是跨平台的， sun公司针对不同的操作系统分别设计不同的JVM,来屏蔽不同的系统指令集差异而对外体提供统一的接口(java API)。

## 1.2 JDK, JRE和JVM的区别与联系

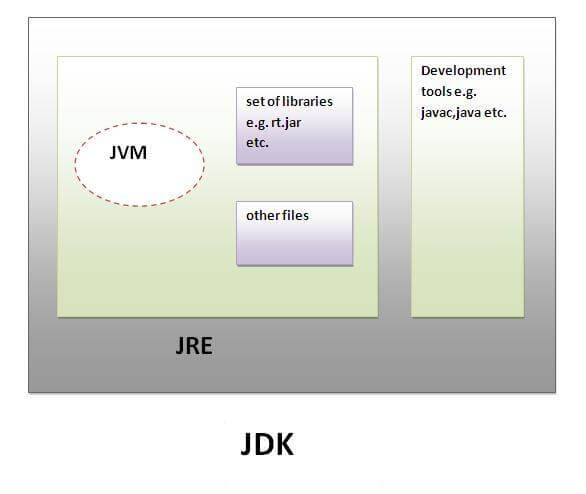
JDK: java development kit, java开发工具包，针对开发者，里面主要包含了jre, jvm, jdk源码包，以及bin文件夹下用于开发，编译运行的一些指令器。

JRE: java runtime environment, java运行时环境，针对java用户，也就是拥有可运行的.class文件包（jar或者war）的用户。里面主要包含了jvm和java运行时基本类库（rt.jar）。rt.jar可以简单粗暴地理解为：它就是java源码编译成的jar包（解压出来看一下），用eclipse开发时，当你ctrl点击发现不能跳转到源文件时，需要把rt.jar对应的源码包加进来，而这里的源码包正是jdk文件夹下的src.zip。

JVM: java虚拟机，将它理解为可以识别class文件的一个小型系统，class文件直接和它交互，所以它让class文件和用户真实的操作系统隔离，屏蔽了用户系统的差异性，给人一种感觉就是java出道的最大特点：一次编译，处处运行（跨平台）。而这也正是java跨平台引起过争议的一点，曾经一些人觉得java跨平台是个幌子，因为前提是用户机子上必须安装了JVM（准确地说是JRE）。

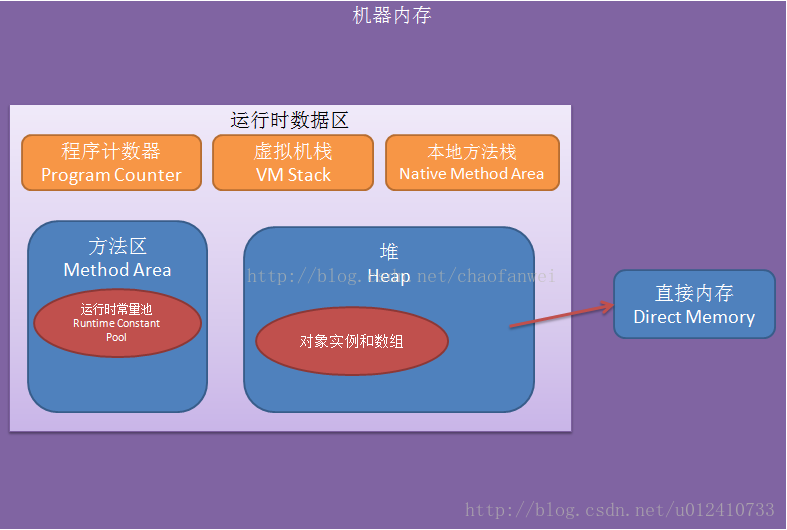
JDK和JRE区别：去bin文件夹下你会发现，JDK有javac.exe而JRE里面没有，众所周知javac指令是用来将java文件编译成class文件的，这是你开发去做的事，用户是不会去做的。JDK还有jar.exe, javadoc.exe等等用于开发的可执行指令文件。这也证实了一个是开发环境，一个是运行环境。

JRE和JVM区别：有些人觉得，JVM就可以执行class了，其实不然，JVM执行.class还需要JRE下的lib类库的支持，尤其是rt.jar。



## 1.3java 内存模型

### **1.内存模型图**



### 2.模型说明

JVM的内存空间分为五个部分，分别是：

① 程序计数器

② Java虚拟机栈

③ 本地方法栈

④ 堆

⑤ 方法区

#### 1.程序计数器

##### 1.1什么是程序计数器

程序计数器是一块较小的内存空间，可以把它看作当前线程正在执行的字节码的行号指示器。也就是说，程序计数器里面记录的是当前线程正在执行的那一条字节码指令的地址。

注：但是，如果当前线程正在执行的是Native方法，那么此时程序计数器为空。

##### 1.2.程序计数器的作用

程序计数器有两个作用

① 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，

如：顺序执行、选择、循环、异常处理。

② 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

##### 1.3.程序计数器的特点

① 是一块较小的存储空间

② 线程私有，每条线程都有一个程序计数器。

③ 是唯一一个不会出现OutOfMemoryError的内存区域。

④ 生命周期随着线程的创建而创建，随着线程的结束而死亡。

#### 2.Java虚拟机栈

##### 2.1什么是Java虚拟机栈

① 重点是Java虚拟机栈，它是线程私有的，生命周期与线程相同。

② 每个方法执行都会创建一个栈帧（是用于支持虚拟机进行方法调用和方法执行的数据结构），用于存放局部变量表，操作数栈，动态链接，方法出口等。每个方法从被调用，直到被执行完，对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

③ 通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，以及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

##### 2.2.Java虚拟机栈的特点

① 局部变量表的创建是在方法被执行的时候，随着栈帧的创建而创建。

而且，局部变量表的大小在编译时期就确定下来了，在创建时只需分配实现规定好大小。

此外，在方法运行的过程局部变量表的大小是不会发生改变的。

② 会有两种异常StackOverFlowError和OutOfMemoryError。

当线程请求栈深度大于虚拟机所允许的深度就会抛出StackOverFlowError错误；

虚拟机栈动态扩展，当扩展无法申请到足够的内存空间时，抛出OutOfMemoryError。

③ Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，

而且随着线程的创建而创建，随着线程的死亡而死亡。

#### 3.本地方法栈

##### 3.1.什么是本地方法栈

①本地方法栈和Java虚拟机栈实现的功能类似，但本地方法区是本地方法运行的内存模型。

②本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

③方法执行完毕后相应的栈帧也会出栈并释放内存空间。也会抛出StackOverFlowError和OutOfMemoryError异常。

#### 4.堆

##### 4.1.什么是堆

堆是用来存放对象的内存空间，几乎所有的对象都存储在堆中。

##### 4.2.堆的特点

① 线程共享

整个Java虚拟机只有一个堆，所有的线程都访问同一个堆。

而程序计数器、Java虚拟机栈、本地方法栈都是一个线程对应一个的。

② 在虚拟机启动时创建

③ 垃圾回收的主要场所

④ 可以进一步细分为：新生代、老年代。

新生代又可被分为：Eden、From Survivor、To Survivor。

不同的区域存放具有不同声明周期的对象，这样可以根据不同的区域使用不同的垃圾回收算法，从而更具有针对性，从而更高效。

⑤ 堆的大小既可以固定也可以扩展，但主流的虚拟机堆的大小是可扩展的，

因为当线程请求分配内存，但堆已满，且内存已满无法再扩展时，就抛出OutOfMemoryError

#### 5.方法区

##### 5.1.什么是方法区

Java虚拟机规范中定义方法区是堆的一个逻辑部分。

方法区中存放已经奔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。

##### 5.2.方法区的特点

① 线程共享

方法区是堆的一个逻辑部分，因此都是线程共享的，整个虚拟机中只有一个方法区。

② 永久代

方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，我们把方法区称为永久代。

③ 内存回收效率低

方法区中的信息一般需要长期存在，回收一遍内存之后可能只有少量信息无效。

对方法区的内存回收的主要目标是：对常量池的回收 和 对类型的卸载。

④ Java规范对方法区的要求比较宽松

和堆一样，允许固定大小，也允许可扩展的大小，还允许不实现垃圾回收

##### 5.3.什么是运行时常量池

方法区中存放三种数据：类信息、常量、静态变量、即时编译器编译后的代码。

其中常量存储在运行时常量池中。

我们一般在一个类中通过public static final来声明一个常量。

这个类被编译后便生成Class文件，这个类的所有信息都存储在这个Class文件中。

当这个类被Java虚拟机加载后，class文件中的常量存放在方法区的运行时常量池中。而且在运行期间，可以向常量池中添加新的常量。

如：String类中的intern()方法就能在运行期间向常量池中添加字符串常量。

当运行时常量池中的某些常量没有被对象引用，同时也没有被变量引用，

那么就需要垃圾收集器回收。

#### 6.直接内存

直接内存是除了Java虚拟机之外的内存，但也有可能被Java使用。

在NIO中引入了一种基于通道和缓冲的IO方式。它可以通过调用本地方法直接分配Java虚拟机之外的内存，然后通过一个存储在Java堆中的DirectByteBuffer对象直接操作该内存，而无需先将外面内存中的数据复制到堆中再操作，从而提升了数据操作的效率。

直接内存的大小不受Java虚拟机控制，但既然是内存，当内存不足时就会抛出OOM异常。

#### 7总结

① JVM的内存模型中一共有两个“栈”，分别是：Java虚拟机栈和本地方法栈

两个“栈”的功能类似，都是方法运行过程的内存模型，

并且两个“栈”内部构造相同，都是线程私有。

② Java虚拟机的内存模型中一共有两个“堆”，一个是原本的堆，一个是方法区。

方法区本质上是属于堆的一个逻辑部分，堆中存放对象，方法区中存放类信息、常量、静态变量、即时编译器编译的代码。

③ 堆是Java虚拟机中最大的一块内存区域，也是垃圾收集器主要的工作区域。

④ 程序计数器、Java虚拟机栈、本地方法栈是线程私有的，即每个线程都拥有各自的程序计数器、Java虚拟机栈、本地方法区。并且他们的生命周期和所属线程一样。

而堆、方法区是线程共享的，在Java虚拟机中只有一个堆、一个方法区。

并在JVM启动的时候就创建，JVM停止才销毁。

## 1.4jvm 类加载过程

加载->链接(验证+准备+解析)->初始化->使用->卸载

### 1.4.1加载

在加载阶段，虚拟机需要完成以下3件事情：

① 通过一个类的全限定名来获取定义此类的二进制字节流。

② 将这个字节流所代表的静态存储结果转化为方法区的运行时数据结构。

③ 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

虚拟机规范中并没有准确说明二进制字节流应该从哪里获取以及怎样获取，

这里可以通过定义自己的类加载器去控制字节流的获取方式。

### 1.4.2链接

#### 1.链接-验证

为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，

并且不会危害虚拟机自身的安全。

包括4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

#### 2.链接-准备

正式为类变量(static修饰的字段变量)分配内存并且设置该类变量的初始值即0(static int i = 5;这里只将i初始化为0，至于5的值将在初始化赋值)，

这里不包括用final修饰的static，因为final在编译的时候就会分配了，

注意这里不会为实例变量分配初始化，类变量会分配在方法区中，

而实例变量是会随着对象一起分配到Java堆中。

#### 3.链接-解析

主要将常量池中的符号引用替换为直接引用的过程。

符号引用就是一组符号来描述目标，可以是任何字面量，

而直接引用就是指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

有类或接口的解析，字段的解析，类方法解析，接口方法解析。

### 1.4.3初始化

#### 1.类什么时候初始化

① 创建类的实例，也就是new一个对象

② 访问某个类或接口的静态变量，或者对该静态变量赋值

③ 调用类的静态方法

④ 反射(Class.forName(“com.grace.load”))

⑤ 初始化一个类的子类(会首先初始化子类的父类)

⑥ JVM启动时标明的启动类，即文件名和类名相同的那个类

#### 2.类的初始化顺序

① 如果这个类还没有被加载和链接，那先进行加载和链接

② 加入这个类存在直接父类，并且这个类还没有被初始化

(注意：在一个类加载器中，类只能初始化一次)，那就初始化直接的父类(不适用于接口)

③ 类中存在初始化语句(如static变量和static块)，那就依次执行这些初始化语句。

④ 总得来说，初始化顺序依次是：

(静态变量、静态初始化块) -> (变量、初始化块) -> 构造器

如果有父类，则顺序是：

父类static方法 -> 子类static方法 -> 父类构造方法 -> 子类构造方法

#### 3.《深入JVM补充》

对于初始化阶段，虚拟机规范则是严格规定了有且只有5种情况必须立即对类进行“初始化”

① 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，

如果类没有进行过初始化，则需要先触发其初始化。

生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、

读取或设置一个类的静态字段(被final修饰、已在编译期把结果放入常量池的静态字段除外)的时候，以及调用一个类的静态方法的时候。

② 使用java.lang.reflect包的方法对类进行反射调用的时候，

如果类没有进行过初始化，则需要先触发其初始化。

③ 当初始化一个类的时候，如果发现其父类还没有进行过初始化，

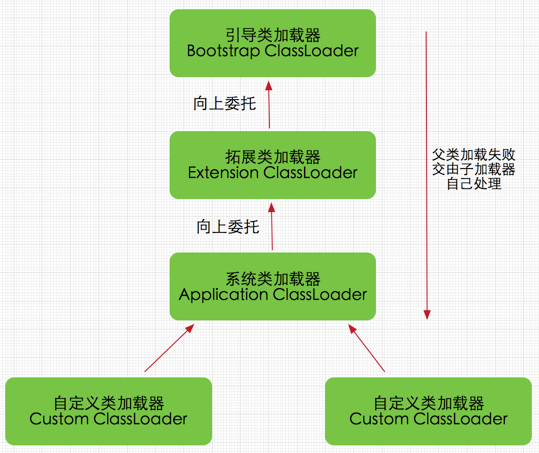
则需要先触发其父类的初始化。

④ 当虚拟机启动时，用户需要指定一个要执行的主类(包含main()方法的那个类)， 虚拟机会先初始化这个主类。

⑤当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

### 1.4.4Java的类加载器都有哪些

#### 1.什么是双亲委派模型



双亲委派模型要求除了顶层的启动类加载器外，其余类加载器都应当有自己的父类加载器。

这里的类加载器之间的父子关系一般不会以继承的关系来实现，而是都使用组合关系来复用父加载器的代码。

双亲委派模型的工作过程是：

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求(它得搜索范围中没有找到所需的类)时，子加载器才会尝试自己去加载。

#### 2.双亲委派模式优势

采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，

因此Object类在程序的各种类加载器环境中都是同一个类。

相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，

如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，

那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

#### 3.启动类加载器(Bootstrap ClassLoader)

将存放于<JAVA\_HOME>\lib目录中，或者被-Xbootclasspath参数所指定的路径中的，

并且是虚拟机识别的(仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载)类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，

那直接使用null代替即可。

#### 4.扩展类加载器(Extension ClassLoader)

这个加载器由sun.misc.Launcher$ExtClassLoader实现，

它负责加载<JAVA\_HOME>\lib\ext目录中的，

或者被java.ext.dirs系统变量所指定的路径中的所有类库，

开发者可以直接使用扩展类加载器。

#### 5.应用程序类加载器(Application ClassLoader)

这个类加载器由sun.misc.Launcher$AppClassLoader实现。

由于这个类加载器时ClassLoader中getSystemClassLoader()方法的返回值，

所以一般也称它为系统类加载器。

它负责加载用户类路径(ClassPath)上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它是一种任务委派模式。

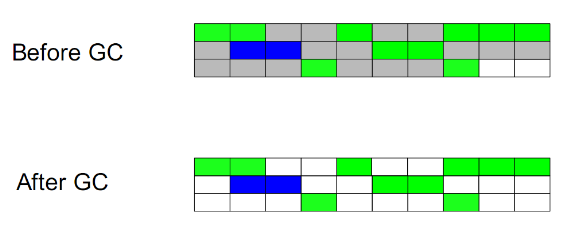
## 1.5****垃圾回收算法****

### 1.5.1按照基本回收策略分

#### 1引用计数（Reference Counting）

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

#### 2标记-清除（Mark-Sweep）

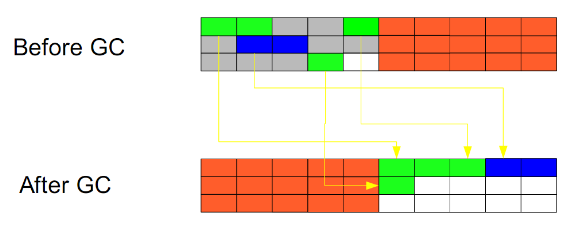


此算法执行分两阶段:

第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。

此算法需要暂停整个应用，同时，会产生内存碎片。

#### 3复制（Copying）



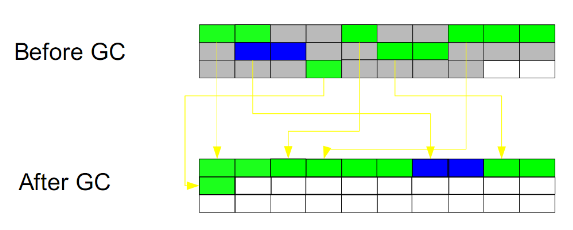
此算法把内存空间划分为两个相等的区域，每次只是用其中一个区域。

垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。

此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。

当然，此算法的缺点也是比较明显的，就是需要两倍内存空间。

#### 4标记-整理（Mark-Compact）



此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

### 1.5.2按分区对待的方式分

#### 1. ****增量收集（Incremental Collecting）****

实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

#### 2. 分代收集（Generational Collecting）

基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

### 1.5.3按系统线程分

#### 1. ****串行收集****

串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。

#### 2. ****并行收集****

并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上CPU数目越多，越能体现出并行收集器的优势。

#### ****并发收集****

GC线程和应用线程大部分时间是并发执行，只是在初始标记（initial mark）和二次标记（remark）时需要stop-the-world，这可以大大缩短停顿时间（pause time），所以适用于响应时间优先的应用，减少用户等待时间。

由于GC是和应用线程并发执行，只有在多CPU场景下才能发挥其价值，在执行过程中还会产生新的垃圾floating garbage， 如果等空间满了再开始GC，那这些新产生的垃圾就没地方放了，这时就会启动一次串行GC，等待时间将会很长，所以要在空间还未满时就要启动GC。

mark和sweep操作会引起很多碎片，所以间隔一段时间需要整理整个空间，否则遇到大对象，没有连续空间也会启动一次串行GC。

采用此收集器（如tenured generation），收集频率不能大，否则会影响到cpu的利用率，进而影响吞吐量。

#### 4. ****回收器小结****

    串行处理器：

         -- 适用情况：数据量比较小（100M左右），单处理器下并且对响应时间无要求的应用。

         -- 缺点：只能用于小型应用。

      并行处理器：

          -- 适用情况：“对吞吐量有高要求”，多CPU，对应用响应时间无要求的中、大型应用。举例：后台处理、科学计算。

          -- 缺点：垃圾收集过程中应用响应时间可能加长。

      并发处理器：

          -- 适用情况：“对响应时间有高要求”，多CPU，对应用响应时间有较高要求的中、大型应用。举例：Web服务器/应用服务器、电信交换、集成开发环境。

## 1.6 ****GC收集器有哪些？CMS收集器与G1收集器的特点。****

1.Serial色儿瑞哦（复制算法）：一个单线程的收集器，在进行垃圾收集时候，必须暂停其他所有的工作线程直到它收集结束。

特点：CPU利用率最高，停顿时间即用户等待时间比较长。

2.Parallel（复制算法）：采用多线程来通过扫描并压缩

特点：停顿时间短，回收效率高，对吞吐量要求高。

3.CMS收集器（标记-清除）：采用“标记-清除”算法实现，使用多线程的算法去扫描堆，对发现未使用的对象进行回收。

4:G1（标记整理）:堆被划分成 许多个连续的区域(region)。采用G1算法进行回收，吸收了CMS收集器特点。

特点：支持很大的堆，高吞吐量、支持多CPU和垃圾回收线程、在主线程暂停的情况下，使用并行收集、在主线程运行的情况下，使用并发收集

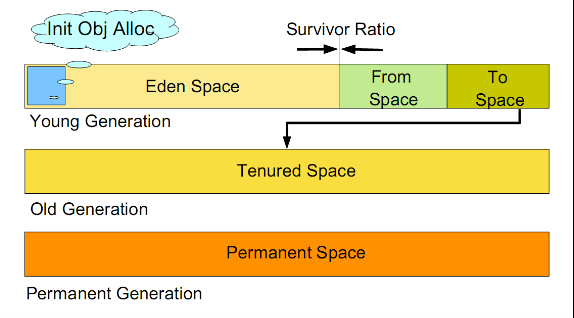
## 1.7为什么要采用分代收集

 分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的。因此，不同声明周期的对象可以采取不同的收集方式，以便提高回收效率。**

    在**Java**程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如Http请求中的Session对象、线程、Socket连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期比较短，比如：String对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

        是想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是它们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

## 1.8分代收集如何分代



如上图所示  
虚拟机中的共划分为三个代：**年轻代（Young Generation）**、**年老代（Old Generation）**和**持久代（Permanent Generation）**。

其中持久代主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的

### 1.8.1年轻代

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分为三个区。一个Eden区，两个Survivor区（一般而言）。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活将被复制到另外一个Survivor区，当这个Survivor区也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区（Tenured）”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来的对象和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor区过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

### 1.8.2****年老代****

在**年轻代**中经历了N次垃圾回收后仍然存活的对象，就会被放到**年老代**中。因此，可以认为**年老代**中存放的都是一些生命周期较长的对象。

### 1.8.3 ****持久代****

用于存放静态文件，如**java**类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class,例如Hibernate等，在这种时候需要设置一个比较大的持久空间来存放这些运行过程中新增的类。持久代大小通过 -XX:MaxPermSize = <N> 进行设置。

## 1.9 Minor GC、Major GC和Full GC之间的区别

GC是负责回收所有无任何引用对象的内存空间。 注意:垃圾回收回收的是无任何引用的对象占据的内存空间而不是对象本身

### 1.9.1 Minor GC

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC

特点：

1. 当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 Eden 区满了。所以分配率越高，越频繁执行 Minor GC。
2. 执行 Minor GC 操作时，不会影响到永久代。从永久代到年轻代的引用被当成 GC roots，从年轻代到永久代的引用在标记阶段被直接忽略掉。
3. 质疑常规的认知，所有的 Minor GC 都会触发“全世界的暂停（stop-the-world）”，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。其中的真相就 是，大部分 Eden 区中的对象都能被认为是垃圾，永远也不会被复制到 Survivor 区或者老年代空间。如果正好相反，Eden 区大部分新生对象不符合 GC 条件，Minor GC 执行时暂停的时间将会长很多。

触发机制：当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。

### 1.9.2Major GC、Full GC

**Major GC**：清理老年代

**Full GC**：清理整个堆空间—包括年轻代和老年代

**Major GC / Full GC**：老年代 GC（Major GC / Full GC）：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程） 。MajorGC 的速度一般会比 Minor GC 慢 10倍以上。

因此应该尽可能减少 Full GC 的次数。在对JVM调优的过程中，很大一部分工作就是对于 Full GC 的调节。

**有如下原因可能导致Full GC:**

* 年老代（Tenured）被写满
* 持久代（Perm）被写满
* System.gc()被显式调用
* 上一次GC之后Heap的各域分配策略动态变化

## 1.10 强引用、软引用、弱引用、虚引用与GC的关系

强引用：new出的对象之类的引用，只要强引用还在，永远不会回收。

软引用：引用但非必须的对象，内存溢出异常之前回收。

弱引用：非必须的对象，对象只能生存到下一次垃圾收集发生之前。

虚引用：对生存时间无影响，在垃圾回收时得到通知。

# java基础知识

## 2.1 java 面向对象的特征有那些

有四大基本特征:封装、抽象、继承、多态

1封装：

主要是把过程和数据包围起来，不对外部公开内部的数据和逻辑，从而保护内部的数据结构不被外界改变，起到保护作用！

封装的优点：使对象以外的部分不能随意存取对象内部的数据，有效避免交叉感染，使软件错误局部化，减少查错和排错的难度。

2 继承：

继承是一个层次结构，子类继承父类的方法可以重写或重载！

  使用一个特殊类拥有其全部的属性和服务，作为父类，他可以有多个子类，一般的类去实现，比如：父类为 人，子类可以是 男人和女人，男和女都继承于父类，但是一个子类只能继承一个特殊类，不能继承多个。但一个一般类可以实现多个接口。子类继承父类的时候，可以重写父类的方法，也可以有自己的方法。

3 抽象：

抽象就是找出一些事物的相似和共性之处，然后将这些事物归为一个类，这个类只考虑这些事物的相似和共性之处就是把现实生活中的对象，抽象为类。

Java中抽象可以是抽象类、抽象方法，使用abstract修饰。其中抽象类中可以有抽象方法和非抽象方法，抽象方法。据有抽象方法的类一定是抽象类。抽象方法一般只有方法名，没有方法的实现。如：public abstract void test(); 且抽象方法的权限修饰必须是protected public 的，因为子类在继承抽象父类的时候，必须要实现父类所有的抽象方法。

4 多态：

不同类的对象对同一个类的对象做出不同的响应信息！

具有三个条件：1）继承 2）重写 3）父类对象引用指向子类对象

多态优点：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。

2.可扩充性（extensibility）。多态对代码具有可扩充性。

3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。

5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程。

## 2.2说一下"=="和equals方法究竟有什么区别？

==用来判断两个变量之间的的值是否相等。变量就可以分为基本数据类型变量，引用类型。

如果是基本数据类型的变量直接比较值而引用类型要比较对应的引用的内存的首地址。

equals 用来比较两个对象长得是否一样。判断两个对象的某些特征是否一样。实际上就是调用对象的equals方法进行比较。

## 2.3[String的Intern方法详解](https://www.cnblogs.com/wxgblogs/p/5635099.html)

在 JAVA 语言中有8中基本类型和一种比较特殊的类型String。这些类型为了使他们在运行过程中速度更快，更节省内存，都提供了一种常量池的概念。常量池就类似一个JAVA系统级别提供的缓存。8种基本类型的常量池都是系统协调的，String类型的常量池比较特殊。它的主要使用方法有两种：

* 直接使用双引号声明出来的String对象会直接存储在常量池中。
* 如果不是用双引号声明的String对象，可以使用String提供的intern方法。intern 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中

## 2.4讲一下String和StringBuilder的区别(final)？StringBuffer和StringBuilder的区别?

1. String是内容不可变的字符串。String底层使用了一个不可变的字符数组(final char[])

2. StringBuillder StringBuffer,是内容可以改变的字符串。StringBuillder StringBuffer底层使用的可变的字符数组（没有使用final来修饰）

3. StringBuilder是线程不安全的，效率较高.而StringBuffer是线程安全的，效率较低。

## 2.5讲一下java中的集合?

Java集合大致可以分为Set、List、Queue和Map四种体系，其中Set代表无序、不可重复的集合；List代表有序、重复的集合；而Map则代表具有映射关系的集合，Java 5 又增加了Queue体系集合，代表一种队列集合实现。  
Java集合就像一种容器，可以把多个对象（实际上是对象的引用，但习惯上都称对象）“丢进”该容器中。从Java 5 增加了泛型以后，Java集合可以记住容器中对象的数据类型，使得编码更加简洁、健壮。

## 2.6.ArrayList、LinkedList、Vector的区别

ArrayList和Vector类都是基于数组实现的List类。

ArrayList和Vector的区别

1.ArrayList是线程不安全的，Vector是线程安全的。  
2.Vector的性能比ArrayList差。

LinkedList使用的是链表实现的List类,LinkedList也是非线程安全的。

如果涉及到“动态数组”、“栈”、“队列”、“链表”等结构，应该考虑用List，具体的选择哪个List，根据下面的标准来取舍。  
(01) 对于需要快速插入，删除元素，应该使用LinkedList。  
(02) 对于需要快速随机访问元素，应该使用ArrayList。  
(03) 对于“单线程环境” 或者 “多线程环境，但List仅仅只会被单个线程操作”，此时应该使用非同步的类(如ArrayList)。对于“多线程环境，且List可能同时被多个线程操作”，此时，应该使用同步的类(如Vector)。

## 2.7 HashMap与Hashtable，****ConcurrentHashMap****的区别？

在JDK1.8之前，HashMap采用数组+链表实现，而JDK1.8中，HashMap采用数组+链表+红黑树实现，当链表长度超过阈值(8)时，将链表转换为红黑树，这样大大减少了查找时间。

**相同点**：HashMap和HasheTalbe都可以使用来存储key--value的数据。

**区别**：

1.HashMap是可以把null作为key或者value的，而HashTable是不可以的。

2.HashMap是线程不安全的，效率较高。而HashTalbe是线程安全的，效率较低。

3. Hashtable的初始容量与HashMap不同，Hashtable的初始容量为11，HashMap的初始容量是16

如果又想线程安全但是我又想效率高？使用ConcurrentHashMap

通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)

Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

## 2.8 java IO 流的理解

流是一组有顺序的，有起点和终点的字节集合，即数据在两设备间的传输称为流。

### 2.8.1 IO流的分类

① 根据处理数据类型的不同分为：字符流和字节流

② 根据数据流向不同分为：输入流和输出流

### 2.8.2字符流和字节流

字节流和字符流的区别：

① 读写单位不同：字节流以字节(8bit)为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。

② 处理对象不同：字节流能处理所有类型的数据(如图片、avi等)，而字符流只能处理字符类型的数据。

③ 字节流：一次读入或读出是8位二进制

④ 字符流：一次读入或读出是16位二进制

结论：只要是处理纯文本数据，就优先考虑使用字符流。除此之外都使用字节流。

### 2.8.3.输入流和输出流

输入流只能进行读操作，输出流只能进行写操作，程序中需要根据待传输数据的不同特性而使用不同的流。

### 2.8.4输入字节流InputStream

① InputStream是所有的输入字节流的父类，它是一个抽象类。

② ByteArrayInputStream、StringBufferInputStream、FileInputStream

是三种基本介质流，他们分别从Byte数组、StringBuffer和本地文件中读取数据。

② PipeInputStream是从与其他线程共用的管道中读取数据。

③ ObjectInputStream和FilterInputStream的子类都是装饰流(装饰器模式的主角)

### 2.8.5输出字节流(OutputStream)

① OutputStream是所有的输出字节流的父类，它是一个抽象类。

② ByteArrayOutputStream、FileOutStream是两种基本的介质流，

它们分别向Byte数组和本地文件中写入数据。

③ PipeOutputStream是向与其它线程共用的管道中写入数据。

④ ObjectOutputStream和FilterOutputStream的子类都是装饰流。

总结：

① 输入流：InputStream或者Reader：从文件中读到程序中

② 输出流：OutputStream或者Writer：从程序中输出到文件中

### 2.8.6节点流

节点流：直接与数据源相连，读入或读出

直接使用节点流，读写不方便，为了更快的读写文件，才有了处理流

常用的节点流

① 父类：InputStream、OutStream、Reader、Writer等

② 文件：FileInputStream、FileOutStream、FileReader、FileWriter文件进行处理的节点流。

③ 数组：ByteArrayInputStream、ByteArrayOutputStream、CharArrayReader、CharArrayWriter对数组进行处理的节点流(对应的不再是文件，而是内存中的一个数组)

④ 字符串：StringReader、StringWriter对字符串进行处理的节点流

⑤ 管道：PipedInputStream、PipedOutputStream、PipedReader、PipedWriter对管道进行处理的节点流。

### 2.8.7处理流

处理流和节点流一块使用，在节点流的基础上，再套接一层，套接在节点流上的就是处理流。如BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数，

一个流对象经过其他流的多次包装，称为流的链接。

常用的处理流

① 缓冲流：BufferedInputStream、BufferedOutputStream、BufferedReader、BufferedWriter增加缓冲功能，避免频繁读写硬盘。

② 转换流：InputStreamReader、OutputStreamReader实现字节流和字符流之间的转换。

③ 数据流：DataInputStream、DataOutputStream等提供将基础数据类型写入到文件中，或者读取出来。

### 2.8.8 NIO 和普通IO 的区别

NIO即New IO，这个库是在JDK1.4中才引入的。NIO和IO有相同的作用和目的，但实现方式不同，NIO主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

下表总结了Java IO和NIO之间的主要区别：

|  |  |
| --- | --- |
| **IO** | **NIO** |
| 面向流 | 面向缓冲 |
| 阻塞IO | 非阻塞IO |
| 无 | 选择器 |

## 2.9 java 多线程

### 2.9.1讲一下线程的几种实现方式?启动方式？区分方式？

①实现方式

1. 通过继承Thread类实现一个线程
2. 通过实现Runnable接口实现一个线程

继承扩展性不强，java总只支持单继承，如果一个类继承Thread就不能继承其他的类了。

②怎么启动？

Thread thread = new Thread(继承了Thread的对象/实现了Runnable的对象)

thread.setName(“设置一个线程名称”);

thread.start();

启动线程使用start方法，而启动了以后执行的是run方法。

③怎么区分线程？在一个系统中有很多线程，每个线程都会打印日志，我想区分是哪个线程打印的怎么办？

thread.setName(“设置一个线程名称”); 这是一种规范，在创建线程完成后，都需要设置名称。

### 2.9.2线程安全和线程同步锁的了解

线程不安全

多个线程对同一个对象中的实例变量进行并发访问时发生，产生的后果就是“脏读”，

也就是取到的数据其实是被更改过的。

线程安全

就是以获得的实例变量的值是经过同步处理的，不会出现脏读的现象。

线程同步锁

① synchronized同步方法

关键字synchronized取得的锁都是对象锁，而不是把一段代码或方法(函数)当作锁，

哪个线程先执行带synchronized关键字的方法，哪个线程就持有该方法所属对象的锁，那么其他线程只能呈等待状态，前提是多个线程访问的是同一个对象。

② synchronized同步代码块

用关键字synchronized声明方法在某些情况下是有弊端的，

比如A线程调用同步方法执行一个长时间的任务，那么B线程则必须等待比较长时间。

在这样的情况下可以使用synchronized同步代码块来解决。

synchronized方法是对当前对象进行加锁，

而synchronized代码块是对某一个对象进行加锁。

③ 静态同步synchronized方法

synchronized关键字加到static静态方法上是给Class类上锁，

而synchronized关键字加到非static静态方法上是给对象上锁。

④ 与synchronized(class)代码块

同步synchronized(class)代码块的作用其实和synchronized static方法的作用一样。

|  |
| --- |
| public static void printA() {  synchronized(Service.class) {  }  } |

### 2.9.3 volatile和synchronized区别

首先需要理解线程安全的两个方面：执行控制和内存可见。

执行控制的目的是控制代码执行（顺序）及是否可以并发执行。

内存可见控制的是线程执行结果在内存中对其它线程的可见性。

synchronized关键字解决的是执行控制的问题，它会阻止其它线程获取当前对象的监控锁，这样就使得当前对象中被synchronized关键字保护的代码块无法被其它线程访问，也就无法并发执行。更重要的是，synchronized还会创建一个内存屏障，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中，从而保证了操作的内存可见性

volatile关键字解决的是内存可见性的问题，会使得所有对volatile变量的读写都会直接刷到主存，即保证了变量的可见性。这样就能满足一些对变量可见性有要求而对读取顺序没有要求的需求。

**用volatile的条件有两点：**1.对变量的写操作不依赖于当前值。   
2.该变量没有包含在具有其他变量的不变式中。

**volatile和synchronized的区别**

* volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
* volatile仅能使用在变量级别；synchronized则可以使用在方法、和类级别的
* volatile仅能实现变量的修改可见性，不能保证原子性；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；synchronized可能会造成线程的阻塞。

**原子性表现在：要么不执行，要么执行到底**

### 2.9.4volatile关键字的作用

一个变量如果用volatile修饰了，则Java可以确保所有线程看到这个变量的值是一致的，如果某个线程对volatile修饰的共享变量进行更新，那么其他线程可以立马看到这个更新。

### 2.9.5****说说你对volatile的理解？****

<https://mp.weixin.qq.com/s?__biz=MzU0OTk3ODQ3Ng==&mid=2247484058&idx=1&sn=d5c1533204ea655e65947ec57f924799&chksm=fba6ea99ccd1638f945c585cf3b2df6f4d4112b17ea3648730d50fdb5508555d5f30316f4186&mpshare=1&scene=23&srcid=0121iAqocdCvbf2U7KjVnlET#rd>

### 2.9.6线程池的作用？

1. 限定线程的个数，不会导致由于线程过多导致系统运行缓慢或崩溃
2. 线程池不需要每次都去创建或销毁，节约了资源、响应时间更快。

连接池也是一样。

### 2.9.7 java线程池及其特点

线程不允许使用Executors创建，而是通过ThreadPoolExecutor的方式创建，这样的处理方式能让编写代码的工程师更加明确线程池的运行规则，规避资源消耗的风险。

说明：Executors返回的线程池对象的弊端如下.

1) FixedThreadPool和SingleThreadPool：

允许的请求队列长度为Integer.MAX\_VALUE,可能会堆积大量的请求，从而导致OOM。

2) CachedThreadPool和ScheduledThreadPool：

允许的创建线程数量为Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM。

ThreadPoolExecutor有7个参数：

|  |
| --- |
| **new** ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, threadFactory, handler) |

① corePoolSize

核心线程数，当有任务进来的时候，如果当前线程数还未达到corePoolSize个数，则创建核心线程，核心线程有几个特点：

1)当线程数未达到核心线程最大值的时候，新任务进来，即使有空闲线程，也不会复用，仍然新建核心线程。

2)核心线程一般不会被销毁，即使是空闲的状态，但是如果通过方法allowCoreThreadTimeOut(boolean value)设置为true时，超时也同样会被销毁；

3)生产环境首次初始化的时候，可以调用prestartCoreThread 方法来预先创建所有核心线程，避免第一次调用缓慢。

② maximumPoolSize

除了有核心线程外，有些策略是当核心线程完全无空闲的时候，还会创建一些临时的线程来处理任务，maximumPoolSize就是核心线程+临时线程的最大上限，临时线程有一个超时机制，超过了设置的空闲时间没事干，就会被销毁。

③ keepAliveTime

这个就是上面两个参数里所提到的超时时间，也就是线程的最大空闲时间，默认用于非核心线程，通过allowCoreThreadTimeOut(boolean value)方法设置后，也会用于核心线程。

④ unit

这个参数配合上面的keepAliveTime,指定超时的时间单位，秒、分、时等。

⑤ workQueue

等待执行的任务队列，如果核心线程没有空闲的了，新来的任务就会被放到这个等待队列中，这个参数其实一定程度上决定了线程池的运行策略，为什么这么说呢？因为队列分为有界队列和无界队列。

有界队列：队列的长度有上线，当核心线程满载的时候，新任务进来进入队列，当达到上限，又没有核心线程去及时取走处理，这个时候，就会创建临时线程(警惕临时线程无限增加的风险)

无界队列：队列没有上限的，当没有核心线程空闲的时候，新来的任务可以无止境的向队列中添加，而永远也不会创建临时线程。(警惕任务队列无限堆积的风险)

⑥ threadFactory

线程工厂，主要用来创建线程。

⑦ handler

当没有空闲的线程处理任务，并且等待队列已满(当然这只对有界队列有效)，再有新任务进来的话，就要做一些取舍了，而这个参数就是指定取舍策略的，有下面四种策略可以选择：

1) ThreadPoolExecutor.AbortPolicy:直接抛出异常，这是默认策略

2) ThreadPoolExecutor.DiscardPolicy:直接丢弃任务，但是不抛出异常

3) ThreadPoolExecutor.DiscardOldestPolicy:丢弃队列最前面的任务，然后将新来的任务加入等待队列。

4) ThreadPoolExecutor.CallerRunsPolicy:由线程池所在的线程处理该任务，比如在main函数中创建线程池，如果执行此策略，将有main线程来执行该任务。

#### 1.newCachedThreadPool

① 底层

返回ThreadPoolExecutor实例，corePoolSize为0；

maximumPoolSize为Integer.MAX\_VALUE；

keepAliveTime为60L;

unit为TimeUnit.SECONDS；

workQueue为SynchronousQueue(同步队列)

② 通俗

当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可用线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

③ 适用

执行很多短期异步的小程序或者负载较轻的服务器

#### 2.newFixedThreadPool

① 底层

返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，

corePoolSize为nThread，

maximumPoolSize为nThread；

keepAliveTime为0L(不限时)；

unit为：TimeUnit.MILLISECONDS；

WorkQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

② 通俗

创建可容纳固定数量线程的池子，每个线程的存活时间是无限的，当池子满了就不再添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

③ 适用

执行长期的任务，性能好很多

#### 3.newSingleThreadExecutor

① 底层

FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；

unit为：TimeUnit.MILLISECONDS；

workQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

② 通俗

创建只有一个线程的线程池，且线程的存活时间是无限的；

当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

③ 适用

一个任务一个任务执行的场景

#### 4.newScheduledThreadPool

① 底层

创建ScheduledThreadPoolExecutor实例，

corePoolSize为传递来的参数，

maximumPoolSize为Integer.MAX\_VALUE；

keepAliveTime为0；

unit为：TimeUnit.NANOSECONDS；

workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列

② 通俗

创建一个固定大小的线程池，线程池内线程存活时间无限制，

线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

③ 适用

周期性执行任务的场景

**线程池任务执行流程：**

当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。

当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行。

当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务。

当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理。

当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程

当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

### 2.9.8可重入锁和synchronized的区别

ReenTrantLock可重入锁是在java5提供的锁实现。

**1.性能的区别**

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

**2.功能区别**

便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

**3.ReenTrantLock独有的能力**

1.      ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

2.      ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

3.      ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()

**4.什么情况下使用ReenTrantLock**

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

### 2.9.9简单介绍一下读写锁

读写锁表示也有两个锁，一个是读操作相关的锁，也称为共享锁；另一个是写操作相关的锁，也叫排他锁。也就是多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥。在没有线程Thread进行写入操作时，进行读取操作的多个Thread都可以获取读锁，而进行写入操作的Thread只有在获取写锁后才能进行写入操作。即多个Thread可以同时进行读取操作，但是同一时刻只允许一个Thread进行写入操作。

### 2.9.10 CAS，你懂了吗？

CAS是英文单词CompareAndSwap的缩写，中文意思是：比较并替换。CAS需要有3个操作数：内存地址V，旧的预期值A，即将要更新的目标值B。

CAS指令执行时，当且仅当内存地址V的值与预期值A相等时，将内存地址V的值修改为B，否则就什么都不做。整个比较并替换的操作是一个原子操作。

### 2.9.11常见的并发容器

#### 1.ConcurrentHashMap

ConcurrentHashMap实现了Hashtable的所有功能，线程安全，但却在检索元素时不需要锁定，因此效率更高。

ConcurrentHashMap的key 和 value都不允许null出现。原因在于ConcurrentHashMap不能区分出value是null还是没有map上，相对的HashMap却可以允许null值，在于其使用在单线程环境下，可以使用containKey（key）方法提前判定是否能map上，从而区分这两种情况

#### 2.ConcurrentLinkedQueue

ConcurrentLinkedQueue使用链表作为数据结构，它采用无锁操作，可以认为是高并发环境下性能最好的队列。

ConcurrentLinkedQueue是非阻塞线程安全队列，无界，故不太适合做生产者消费者模式，而LinkedBlockingQueue是阻塞线程安全队列，可以做到有界，通常用于生产者消费者模式。

#### 3.CopyOnWriteArrayList

CopyOnWriteArrayList提供高效地读取操作，使用在读多写少的场景。CopyOnWriteArrayList读取操作不用加锁，且是安全的；写操作时，先copy一份原有数据数组，再对复制数据进行写入操作，最后将复制数据替换原有数据，从而保证写操作不影响读操作。

#### 4.ConcurrentSkipListMap

ConcurrentSkipListMap的实现就是实现了一个无锁版的跳表，主要是利用无锁的链表的实现来管理跳表底层，同样利用CAS来完成替换。

### 2.9.12 面试问题之Java 8如何优化CAS性能？

<https://juejin.im/post/5c062c87e51d451dbc21801b>

### 2.9.13数据库悲观锁和乐观锁的原理及应用

#### 悲观锁

总是假设最坏的情况，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塞直到它拿到锁(共享资源每次只给一个线程使用，其他线程阻塞，用完后再把资源转让给其他线程)。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

#### 乐观锁

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

#### 适用场景

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

总结：

两种各有优缺点，读取频繁使用乐观锁，写入频繁使用悲观锁。

## 2.10 Java序列化和反序列化

对象的序列化：将对象转换为二进制流的过程。

对象的反序列化：将二进制流恢复为对象的过程。

较为常用的有Java本身内置的序列化方式、Hessian、JSON、XML。

## 2.11 ThreadLocal的设计理念与作用

ThreadLocal是一个线程的内部存储类，可以在每个线程的内部存储数据，当某个数据的作用域应该对应线程的时候就应该使用它；

而是当某个很复杂的逻辑下的对象传递，需要在线程这个作用域内贯穿其中，用ThreadLocal可以避免这个创建多个静态类。

当你创建一个ThreadLocal对象后，注意，是一个对象，你在不同线程中去访问它的值是可以不一样的，并且是和线程相关联的。

它的实现原理其实比较简单，每个线程中都会维护一个ThreadLocalMap，当在某个线程中访问时，会取出这个线程自己的Map并且用当前ThreadLocal对象做索引来取出相对应的Value值，从而达到不同线程不同值的效果。

## 2.12 如何看待设计模式，java 常用的设计模式、

设计模式就是经过前人无数次的实践总结出的，大部分设计模式都兼顾了系统的可重用性和可扩展性。

1.单例设计模式

2.工厂设计模式

* 简单工厂模式Simple Factory：不利于产生系列产品；
* 工厂方法模式Factory Method：又称为多形性工厂；
* 抽象工厂模式Abstract Factory：又称为工具箱，产生产品族，但不利于产生新的产品；

Spring Bean的创建是典型的工厂模式

1. 建造模式
2. 观察者模式（有时被称作发布/订阅模式，观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。）
3. 适配器模式
4. 代理模式
5. 装饰模式

## 2.13HashMap的实现原理

在JDK1.8之前，HashMap采用数组+链表实现，而JDK1.8中，HashMap采用数组+链表+红黑树实现，当链表长度超过阈值(8)时，将链表转换为红黑树，这样大大减少了查找时间。

<https://www.cnblogs.com/chengxiao/p/6059914.html#t2>

简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，根据key的hash值来求得对应数组中的位置，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可O(1)；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

## 2.14 HashMap数组扩容为什么是2的次幂

HashMap在进行数组扩容时将老数组中的数据逐个链表地遍历，扔到新的扩容后的数组中，我们的数组索引位置的计算是通过 对key值的hashcode进行hash扰乱运算后，再通过和 length-1进行位运算得到最终数组索引位置。保持2的次幂进行扩容就能保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)

# 3.java WEB方面

## 3.1讲一下http get和post请求的区别?

GET和POST请求都是http的请求方式，用户通过不同的http的请求方式完成对资源（url）的不同操作。GET，POST，PUT，DELETE就对应着对这个资源的查 ，改 ，增 ，删 4个操作,具体点来讲GET一般用于获取/查询资源信息，而POST一般用于更新资源信息

1、Get请求提交的数据会在地址栏显示出来，而post请求不会再地址栏显示出来.

GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，多个参数用&连接；POST提交：把提交的数据放置在是HTTP包的包体中。 因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变

2、传输数据的大小

http Get请求由于浏览器对地址长度的限制而导致传输的数据有限制。而POST请求不会因为地址长度限制而导致传输数据限制。

3、安全性,POST的安全性要比GET的安全性高。由于数据是会在地址中呈现，所以可以通过历史记录找到密码等关键信息。

## 3.2Servlet API中forward() 与redirect()的区别？

1.forward是服务器端的转向而redirect是客户端的跳转。

2.使用forward浏览器的地址不会发生改变。而redirect会发生改变。

3.Forward是一次请求中完成。而redirect是重新发起请求。

4.Forward是在服务器端完成，而不用客户端重新发起请求，效率较高。

## 3.3 JSP和Servlet有哪些相同点和不同点?

JSP是Servlet技术的扩展，所有的jsp文件都会被翻译为一个继承HttpServlet的类。JSP侧重于视图，Servlet主要用于控制逻辑。

## 3.4说一下session和cookie的区别？

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

cookie 和session 的区别：

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能,考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：  
   将登陆信息等重要信息存放为SESSION  
   其他信息如果需要保留，可以放在COOKIE中，比如购物车

# java 框架

## 4.1 Spring框架的IoC，AOP及底层实现机制

IOC

将我们创建的对象的方式反转，以前对象的创建是由我们开发人员自己维护，

包括依赖关系也是自己注入；

使用了Spring之后，对象的创建以及依赖关系可以由Spring完成创建以及注入。

反转控制就是反转了对象的创建方式，从我们自己创建反转给了程序。

IoC容器的初始化过程

资源定位，即定义bean的xml -> 载入 ->IoC容器注册，注册beanDefinition

IoC容器的初始化过程，一般不包含bean的依赖注入的实现，在spring IoC设计中，bean的注册和依赖注入是两个过程，依赖注入一般发生在应用第一次索取bean的时候，但是也可以在xml中配置，在容器初始化的时候，这个bean就完成了初始化。

DI：Dependency Injection依赖注入

实现IOC思想需要DI支持

依赖注入的思想是通过工厂模式+反射+配置文件

在Spring的配置文件中，经常看到如下配置：

|  |
| --- |
| <bean name=*"user"* class=*"com.grace.domain.User"* ></bean> |

通过这样的配置，Spring是通过反射帮我们实例化对象

AOP

面向切面编程，在我们的应用中，经常需要做一些事情，但是这些事情与核心业务无关，比如，要记录所有update \* 方法的执行时间，操作人等信息，记录到日志。

(7) AOP的实现原理 - 代理

第一种：jdk动态代理(优先)

其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理，其核心的两个类是InvocationHandler和Proxy。

第二种：cglib代理

第三方代理技术，cglib代理，可以对任何类生成代理，代理的原理是对目标对象进行继承代理。如果目标对象被final修饰，那么该类无法被cglib代理。

这两种代理的区别是jdk动态代理需要目标对象实现接口，而cglib的动态代理则不需要。

## 4.2spring 的5种注入方式

**1.基于注解注入（@Autowired/ @Resource）**

注解可以用于构造器、字段、方法注入。

@Autowired/ @Resource主要有三个属性值：constructor，byName，byType。

constructor：通过构造方法进行自动注入，spring会匹配与构造方法参数类型一致的bean进行注入，如果有一个多参数的构造方法，一个只有一个参数的构造方法，在容器中查找到多个匹配多参数构造方法的bean，那么spring会优先将bean注入到多参数的构造方法中。

byName：就是通过Bean的id或者name自动注入

byType：byType就是按Bean的Class的类型自动注入

@Resource：java的注解，默认以byName的方式去匹配与属性名相同的bean的id，如果没有找到就会以byType的方式查找，如果byType查找到多个的话，使用@Qualifier注解（spring注解）指定某个具体名称的bean。

@Autowired：spring注解，默认是以byType的方式去匹配类型相同的bean，如果只匹配到一个，那么就直接注入该bean，无论要注入的 bean 的 name 是什么；如果匹配到多个，就会调用 DefaultListableBeanFactory 的 determineAutowireCandidate 方法来决定具体注入哪个bean。

**2.set注入**

**3.构造器注入**

**4.静态工厂方法注入**

**5.实例方法注入**

## 4.3spring 的事物传播特性

1.PROPAGATION\_REQUIRED: 如果存在一个事务，则支持当前事务。如果没有事务则开启  
2. PROPAGATION\_SUPPORTS: 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行  
3. PROPAGATION\_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。  
4. PROPAGATION\_REQUIRES\_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。  
5. PROPAGATION\_NOT\_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务。  
6. PROPAGATION\_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常  
7. PROPAGATION\_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行

Propagation

Required （需要） 如果存在一个事务，则支持当前事务。如果没有事务则开启

Supports （支持） 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行

Mandatory （必要的） 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

required\_new 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

Not\_support 总是非事务地执行，并挂起任何存在的事务。

Never （绝不） 总是非事务地执行，如果存在一个活动事务，则抛出异常

Nested （嵌套的） 如果有就嵌套、没有就开启事务

## 4.4 Spring事务的隔离级别

1. ISOLATION\_DEFAULT： 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.

另外四个与JDBC的隔离级别相对应

2. ISOLATION\_READ\_UNCOMMITTED（读取未提交数据）： 这是事务最低的隔离级别，它充许令外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。(会出现脏读, 不可重复读，幻像读， 基本不使用）

3. ISOLATION\_READ\_COMMITTED（读取已提交数据）： 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据

4.ISOLATION\_REPEATABLE\_READ（可重复读）：在同一事务中多次读取数据时, 能够保证所读数据一样, 也就是后续读取不能读到另一事务已提交的更新数据。这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。（可重复读(会出现幻读)）

5. ISOLATION\_SERIALIZABLE（串行化）

这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

MYSQL: 默认为REPEATABLE\_READ级别

SQLSERVER: 默认为READ\_COMMITTED

## 4.5脏读，不可重复读、幻读。

**脏读** : 一个事务读取到另一事务未提交的更新数据

**不可重复读** : 在同一事务中, 多次读取同一数据返回的结果有所不同, 换句话说,

后续读取可以读到另一事务已提交的更新数据. 相反, "可重复读"在同一事务中多次

读取数据时, 能够保证所读数据一样, 也就是后续读取不能读到另一事务已提交的更新数据

**幻读** : 一个事务读到另一个事务已提交的insert数据

幻读的重点在于新增或者删除 (数据条数变化)。同样的条件, 第1次和第2次读出来的记录数不一样

## 4.6Spring 是如何管理事务的，事务管理机制？

Spring的事务机制包括声明式事务和编程式事务。

编程式事务管理：

Spring推荐使用TransactionTemplate，实际开发中使用声明式事务较多。

声明式事务管理：

将我们从复杂的事务处理中解脱出来，获取连接，关闭连接、事务提交、回滚、异常处理等这些操作都不用我们处理了，Spring都会帮我们处理。

声明式事务管理使用了AOP面向切面编程实现的，本质就是在目标方法执行前后进行拦截。在目标方法执行前加入或创建一个事务，在执行方法执行后，根据实际情况选择提交或是回滚事务。

声明式事务的优缺点：

优点

不需要在业务逻辑代码中编写事务相关代码，只需要在配置文件配置或使用注解（@Transaction），这种方式没有侵入性。

缺点

声明式事务的最细粒度作用于方法上，如果像代码块也有事务需求，只能变通下，将代码块变为方法。

## 4.7 [@Transactional 注解的使用和注意](https://www.cnblogs.com/Loadhao/p/6553606.html)

 在需要事务管理的地方加@Transactional 注解。@Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。

1. 不要在接口上声明@Transactional ，而要在具体类的方法上使用 @Transactional 注解，否则注解可能无效。

2.不要图省事，将@Transactional放置在类级的声明中，放在类声明，会使得所有方法都有事务。故@Transactional应该放在方法级别，不需要使用事务的方法，就不要放置事务，比如查询方法。否则对性能是有影响的。

3.使用了@Transactional的方法，对同一个类里面的方法调用， @Transactional无效。比如有一个类Test，它的一个方法A，A再调用Test本类的方法B（不管B是否public还是private），但A没有声明注解事务，而B有。则外部调用A之后，B的事务是不会起作用的。（经常在这里出错）

4.使用了@Transactional的方法，只能是public，@Transactional注解的方法都是被外部其他类调用才有效，故只能是public。道理和上面的有关联。故在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错，但事务无效。

## 4.8 Spring拦截**器**与过滤器的区别

1.二者适用范围不同。Filter是Servlet规范规定的，只能用于Web程序中，而拦截器既可以用于Web程序，也可以用于Application、Swing程序中

2.规范不同。Filter是在Servlet规范定义的，是Servlet容器支持的，而拦截器是在Spring容器内的，是Spring框架支持的

3.深度不同。Filter只在Servlet前后起作用，而拦截器能够深入到方法前后、异常抛出前后等，拦截器的使用有更大的弹性

## 4.9 SpringMVC的核心类是哪几个?

① 控制器核心类：

org.springframework.web.servlet.DispatcherServlet - 配置web.xml

② 加载配置文件核心类

org.springframework.web.context.ContextLoaderListener - spring的配置文件

③ 处理url映射核心类

org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping - 根据bean的名称请求一个bean spring的配置文件 /abc

④ 处理视图资源核心类：

org.springframework.web.servlet.view.ResourceBundleViewResolver

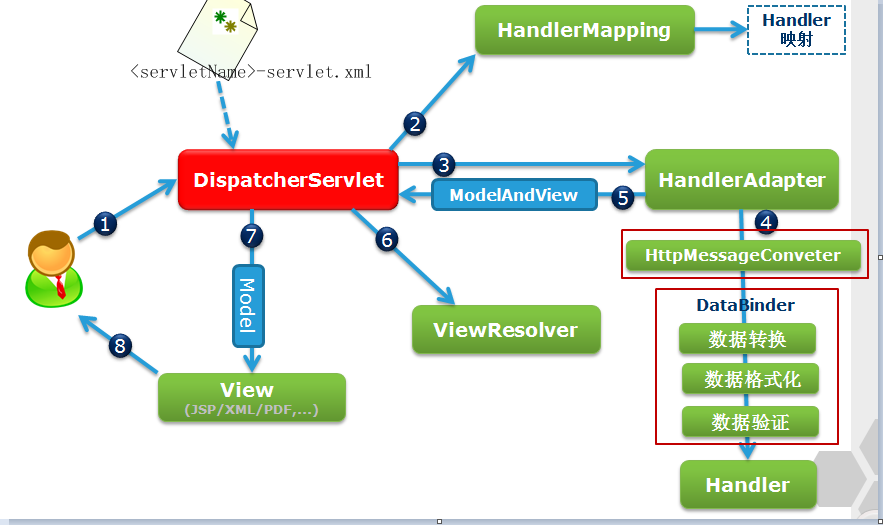
return hello

<context:component-scan base-package="cn.e3mall.service"/>用于替代上面的类

⑥ 方法动态调用核心类

org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver

## 4.10简单讲一下SpringMVC的执行流程？



1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获(捕获)
2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；(查找handler)
3. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller), Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象(执行handler)
4. DispatcherServlet 根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver) (选择ViewResolver)

5、通过ViewResolver 结合Model和View，来渲染视图,DispatcherServlet 将渲染结果返回给客户端。（渲染返回）

快速记忆技巧：

核心控制器捕获请求、查找Handler、执行Handler、选择ViewResolver,通过ViewResolver渲染视图并返回

## 4.11说一下struts2和springMVC有什么不同？

目前企业中使用SpringMvc的比例已经远远超过Struts2,那么两者到底有什么区别，是很多初学者比较关注的问题，下面我们就来对SpringMvc和Struts2进行各方面的比较:

1. 核心控制器（前端控制器、预处理控制器）：对于使用过mvc框架的人来说这个词应该不会陌生，核心控制器的主要用途是处理所有的请求，然后对那些特殊的请求 （控制器）统一的进行处理(字符编码、文件上传、参数接受、异常处理等等),spring mvc核心控制器是Servlet，而Struts2是Filter。

2.控制器实例：Spring Mvc会比Struts快一些（理论上）。Spring Mvc是基于方法设计，而Sturts是基于对象，每次发一次请求都会实例一个action，每个action都会被注入 属性，而Spring MVC更像Servlet一样，只有一个实例，每次请求执行对应的方法即可(注意：由于是单例实例，所以应当避免全局变量的修改，这样会产生线程安全问题)。

3. 管理方式：大部分的公司的核心架构中，就会使用到spring,而spring mvc又是spring中的一个模块，所以spring对于spring mvc的控制器管理更加简单方便，而且提供了全 注解方式进行管理，各种功能的注解都比较全面，使用简单，而struts2需要采用XML很多的配置参数来管理（虽然也可以采用注解，但是几乎没有公司那 样使用）。

4.参数传递：Struts2中自身提供多种参数接受，其实都是通过（ValueStack）进行传递和赋值，而SpringMvc是通过方法的参数进行接收。

5.学习难度：Struts更加很多新的技术点，比如拦截器、值栈及OGNL表达式，学习成本较高，springmvc 比较简单，很较少的时间都能上手。

6.intercepter 的实现机制：struts有以自己的interceptor机制，spring mvc用的是独立的AOP方式。这样导致struts的配置文件量还是比spring mvc大，虽然struts的配置能继承，所以我觉得论使用上来讲，spring mvc使用更加简洁，开发效率Spring MVC确实比struts2高。spring mvc是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应，所以说从架构本身上spring3 mvc就容易实现restful url。struts2是类级别的拦截，一个类对应一个request上下文；实现restful url要费劲，因为struts2 action的一个方法可以对应一个url；而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。spring3 mvc的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架方法之间不共享变量，而struts2搞的就比较乱，虽然方法之间 也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码，读程序时带来麻烦。

1. spring mvc处理ajax请求,直接通过返回数据，方法中使用注解@ResponseBody，spring mvc自动帮我们对象转换为JSON数据。而struts2是通过插件的方式进行处理

## 4.12 什么是ORM?

对象关系映射（Object Relational Mapping，简称ORM）模式是一种为了解决面向**对象**与**关系**数据库存在的互不匹配的现象的技术。简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。

## Hibernate映射对象的状态

临时状态/瞬时状态(transient)：刚刚用new语句创建，没有被持久化

不处于session中(没有使用session的方法去操作临时对象)。该对象成为临时对象

持久化状态/托管状态(persistent)：已经被持久化，加入到session的缓存中。session是没有关闭该状态的对象为持久化对象。

游离状态/脱管状态(detached)：已经被持久化，但不处于session中。

该状态的对象为游离对象。

删除状态(removed)：对象有关联的ID，并且在Session管理下，但是已经被计划(事务提交的时候,commit())删除。如果没有事务就不能删除

## 4.14介绍一下Hibernate的缓存?

Hibernate缓存包括两大类：Hibernate一级缓存和Hibernate二级缓存。

1. Hibernate一级缓存又称为“Session的缓存”。

Session缓存内置不能被卸载，Session的缓存是事务范围的缓存（Session对象的生命周期通常对应一个数据库事务或者一个应用事务）。

一级缓存中，持久化类的每个实例都具有唯一的OID。

1. Hibernate二级缓存又称为“SessionFactory的缓存”。

由于SessionFactory对象的生命周期和应用程序的整个过程对应，因此Hibernate二级缓存是进程范围或者集群范围的缓存，有可能出现并发问题，因此需要采用适当的并发访问策略，该策略为被缓存的数据提供了事务隔离级别。

二级缓存是可选的，是一个可配置的插件，默认下SessionFactory不会启用这个插件。

Hibernate提供了org.hibernate.cache.CacheProvider接口,它充当缓存插件与Hibernate之间的适配器。

缓存命中条件：

一级缓存同一个sessionfactory ,同一个session,oid相同

二级缓存类配置：同一个sessionfactory，OID相同

二级缓存类的集合配置必须配置集合类的二级缓存）命中条件：同一个SessionFactory。OID相同

查询缓存：同一个SessionFactory,发出的sql语句必须相同并且查询的条件也要相同，如果有查询条件，就不能使用查询缓存，命中率非常低

## 4.15 MyBatis与Hibernate有什么不同?

**相同点**：

都是java中orm框架、屏蔽jdbc api的底层访问细节，使用我们不用与jdbc api打交道，就可以完成对数据库的持久化操作。jdbc api编程流程固定，还将sql语句与java代码混杂在了一起，经常需要拼凑sql语句，细节很繁琐。

ibatis的好处：屏蔽jdbc api的底层访问细节；将sql语句与java代码进行分离;提供了将结果集自动封装称为实体对象和对象的集合的功能.queryForList返回对象集合，用queryForObject返回单个对象；提供了自动将实体对象的属性传递给sql语句的参数。

Hibername的好处：Hibernate是一个全自动的orm映射工具，它可以自动生成sql语句，并执行并返回java结果。

**不同点**：

1、hibernate要比ibatis功能强大很多。因为hibernate自动生成sql语句。

2、ibatis需要我们自己在xml配置文件中写sql语句，hibernate我们无法直接控制该语句，我们就无法去写特定的高效率的sql。对于一些不太复杂的sql查询，hibernate可以很好帮我们完成，但是，对于特别复杂的查询，hibernate就很难适应了，这时候用ibatis就是不错的选择，因为ibatis还是由我们自己写sql语句。

ibatis可以出来复杂语句，而hibernate不能。

3、ibatis要比hibernate简单的多。ibatis是面向sql的，不同考虑对象间一些复杂的映射关系。

## 4.16 myBatis 拦截器

MyBatis提供了一种插件(plugin)的功能，虽然叫做插件，但其实这是拦截器功能

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

ParameterHandler (getParameterObject, setParameters)

ResultSetHandler (handleResultSets, handleOutputParameters)

StatementHandler (prepare, parameterize, batch, update, query)

我们看到了可以拦截Executor接口的部分方法，比如update，query，commit，rollback等方法，还有其他接口的一些方法等。

总体概括为：

拦截执行器的方法

拦截参数的处理

拦截结果集的处理

拦截Sql语法构建的处理

## 4.17****说一下spring中Bean的作用域****

**singleton：**

    Spring IoC容器中只会存在一个共享的Bean实例，无论有多少个Bean引用它，始终指向同一对象。Singleton作用域是Spring中的缺省作用域。

**prototype：**

    每次通过Spring容器获取prototype定义的bean时，容器都将创建一个新的Bean实例，每个Bean实例都有自己的属性和状态，而singleton全局只有一个对象。

**request：**

    在一次Http请求中，容器会返回该Bean的同一实例。而对不同的Http请求则会产生新的Bean，而且该bean仅在当前Http Request内有效。

**session：**

    在一次Http Session中，容器会返回该Bean的同一实例。而对不同的Session请求则会创建新的实例，该bean实例仅在当前Session内有效。

**global Session：**

在一个全局的Http Session中，容器会返回该Bean的同一个实例，仅在使用portlet context时有效。

## ****4.18说一下spring中Bean的生命周期****

* 实例化一个Bean，也就是我们通常说的new。
* 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入。
* 如果这个Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的是Spring配置文件中Bean的ID。
* 如果这个Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()，传递的是Spring工厂本身（可以用这个方法获取到其他Bean）。
* 如果这个Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文。
* 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用After方法，也可用于内存或缓存技术。
* 如果这个Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法。
* 如果这个Bean关联了BeanPostProcessor接口，将会调用postAfterInitialization(Object obj, String s)方法。
* 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean接口，会调用其实现的destroy方法。
* 最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法

## 4.19 ****Spring框架中都用到了哪些设计模式？****

* 代理模式：在AOP和remoting中被用的比较多。
* 单例模式：在spring配置文件中定义的bean默认为单例模式。
* 模板方法模式：用来解决代码重复的问题。
* 前端控制器模式：Spring提供了DispatcherServlet来对请求进行分发。
* 依赖注入模式：贯穿于BeanFactory / ApplicationContext接口的核心理念。
* 工厂模式：BeanFactory用来创建对象的实例。

## 4.20 ****BeanFactory 和ApplicationContext的区别****

BeanFactory和ApplicationContext都是接口，并且ApplicationContext是BeanFactory的子接口。

BeanFactory是Spring中最底层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能。而ApplicationContext是Spring的一个更高级的容器，提供了更多的有用的功能。

ApplicationContext提供的额外的功能：国际化的功能、消息发送、响应机制、统一加载资源的功能、强大的事件机制、对Web应用的支持等等。

加载方式的区别：BeanFactory采用的是延迟加载的形式来注入Bean；ApplicationContext则相反的，它是在Ioc启动时就一次性创建所有的Bean,好处是可以马上发现Spring配置文件中的错误，坏处是造成浪费。

# 消息中间件

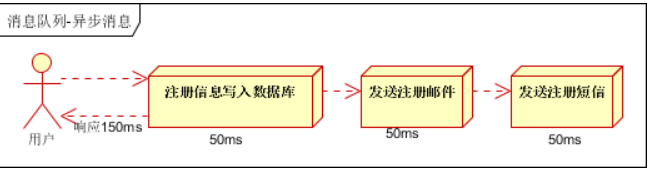
## 5.1消息队列的常用几种介绍下及应用场景

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合、异步消息、流量削峰等问题，实现高性能、高可用、可伸缩和最终一致性架构。使用较多的消息队列有ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ、RocketMQ。

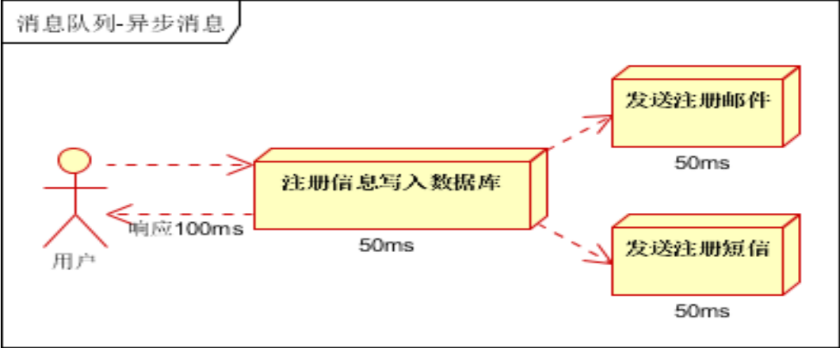
### 5.1.1异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法两种：串行的方式和并行方式。

串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户。



并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端，与串行的差别是，并行的方式可以提高处理的时间。

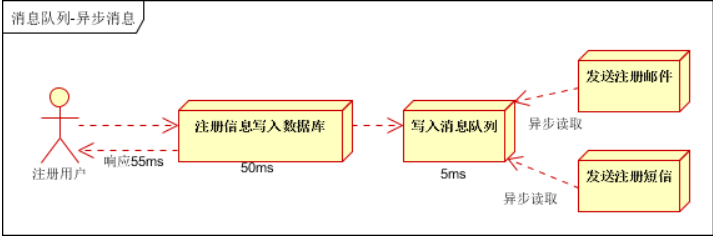


假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150ms，并行的时间可能是100ms。

因为CPU在单位时间内处理的请求是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次(1000/50)。并行方式处理的请求量是10次(1000/1000)。

小结：如以上案例描述，传统的方式系统的性能(并发量，吞吐量，响应时间)会有瓶颈。然后解决这个问题呢？

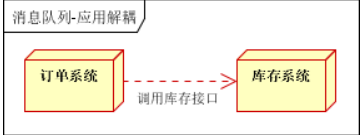
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50ms。因此架构改变后，系统的吞吐量提高到每秒20QPS。比串行提高了3倍，比并行提高了2倍。

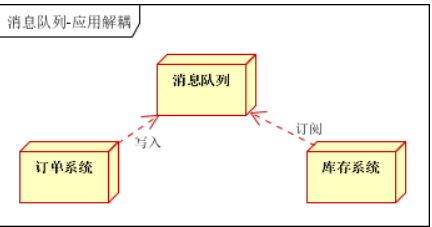
### 5.1.2应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图：



**传统模式的特点：**

假如库存系统无法访问，则订单减存将失败，从而导致订单失败，订单系统与库存系统耦合。如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：

用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功。

库存系统：

订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。

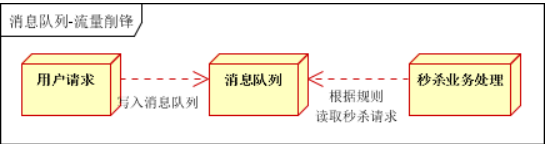
假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

### 5.1.3 流量削峰

流量削峰也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛！

**应用场景**：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

可以控制活动的人数，可以缓解短时间内高流量压垮应用。



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

秒杀业务根据消息队列中的请求信息，再做后续处理。

### 5.1.4日志处理

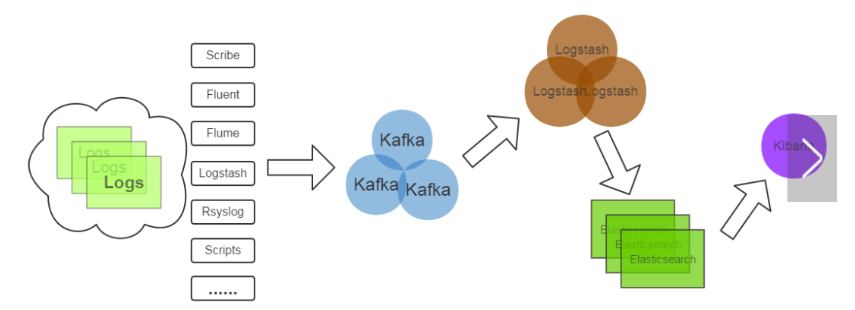
日志处理时指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

架构简化如下：



日志采集客户端，负责日志数据采集，定时写受写入Kafka队列；Kafka消息队列，负责日志数据的接收，存储和转发；日志处理应用;订阅并消费Kafka队列中的日志数据。

下面是新浪kafka日志处理应用案例：



Kafka:接收用户日志的消息队列；

Logstash：做日志解析，同一成JSON输出给Elasticsearch；

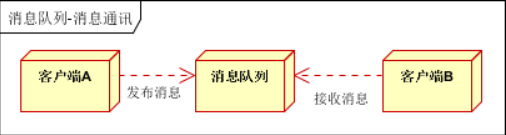
Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能；

kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ElK stack的重要原因。

### 5.1.5消息通讯

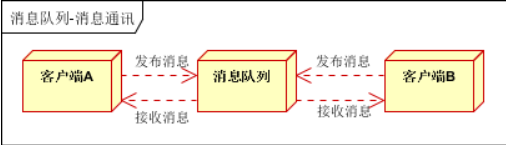
消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

**点对点通讯：**



客户端A和客户端B使用同一队列，进行消息通讯

**聊天室通讯**：



客户端A、客户端B，客户端N订阅同一主题，进行消息发布和接收，实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

## 5.2消息中间件的各种面试题

<https://mp.weixin.qq.com/s?__biz=MzU0OTk3ODQ3Ng==&mid=2247484144&idx=1&sn=a6b86d38a762e317ba78e2500fb1a8ff&chksm=fba6eaf3ccd163e5403a01be51216780040511b2ddc4d5750ace6e46b4242ceaf77d0432c680&mpshare=1&scene=23&srcid=0123LivyvUah7ZSpq3njSWvv#rd>

## 5.3 引入消息中间件，还有分布式一致性的问题。

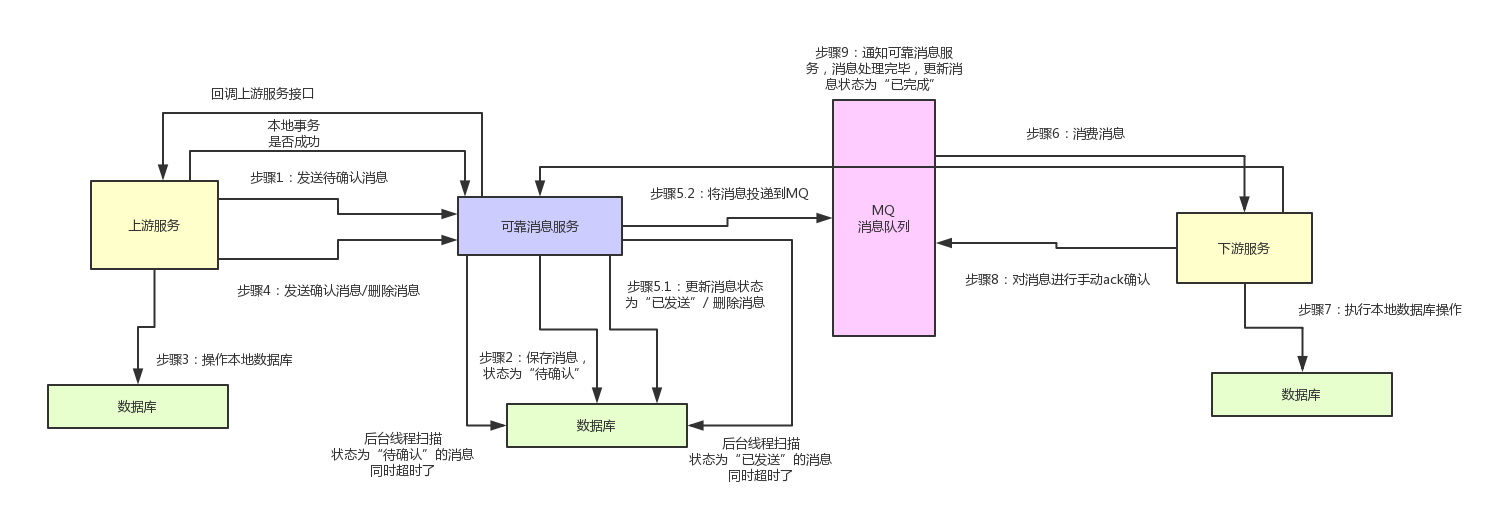
举个例子，比如说系统C现在处理自己本地数据库成功了，然后发送了一个消息给MQ，系统D也确实是消费到了。

但是结果不幸的是，系统D操作自己本地数据库失败了，那这个时候咋办？

系统C成功了，系统D失败了，会导致系统整体数据不一致了啊。

所以此时又需要使用可靠消息最终一致性的分布式事务方案来保障。

解决方案：<https://juejin.im/post/5bf2c6b6e51d456693549af4>



## 5.4 如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

（1）先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉；

（2）新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量；

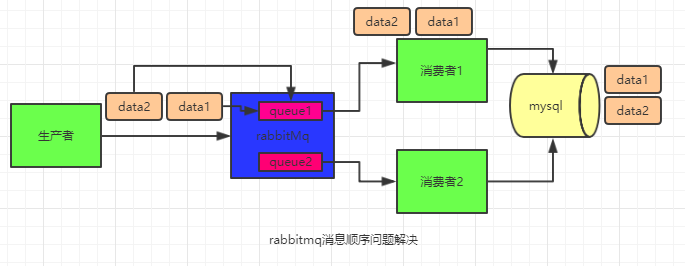
（3）然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue；

（4）接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据；

（5）这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据；

（6）等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息；

## 5.5 [如何保证消息的顺序性？](https://github.com/liuxc1/advanced-java/blob/master/docs/high-concurrency/how-to-ensure-the-order-of-messages.md)



如图，在 MQ 里面创建多个 queue，同一规则的数据（对唯一标识进行 hash），有顺序的放入 MQ 的 queue 里面，消费者只取一个 queue 里面获取数据消费，这样执行的顺序是有序的。或者还是只有一个 queue 但是对应一个消费者，然后这个消费者内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

## 5.6如何保证消息不被重复消费？或者说，如何保证消息消费的幂等性？

回答这个问题，首先你别听到重复消息这个事儿，就一无所知吧，你先大概说一说可能会有哪些重复消费的问题。

**幂等性**，通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

**具体解决方案：**

* 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。
* 比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。
* 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。
* 比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

## 5.7如何保证消息队列的高可用？

### RabbitMQ 的高可用性

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。

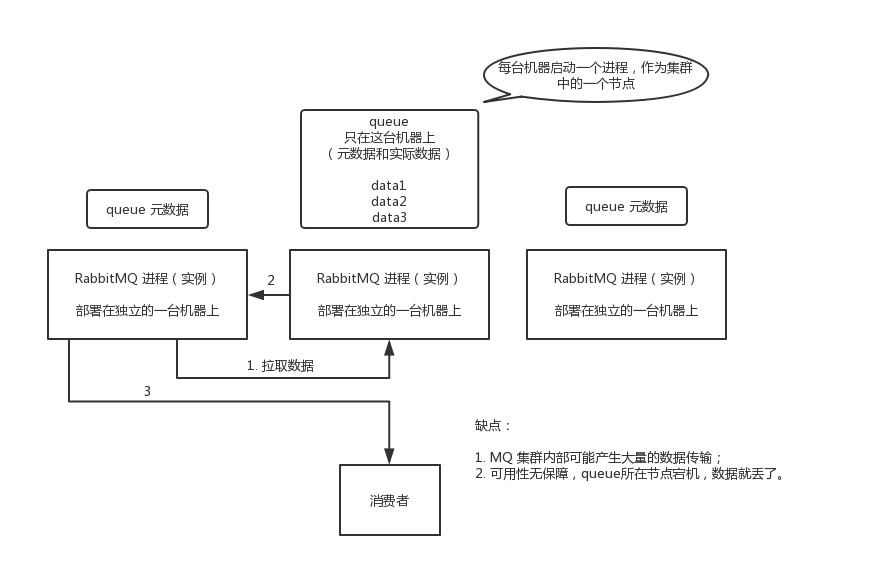
RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

#### 单机模式

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的smile，没人生产用单机模式。

#### 普通集群模式（无高可用性）

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。



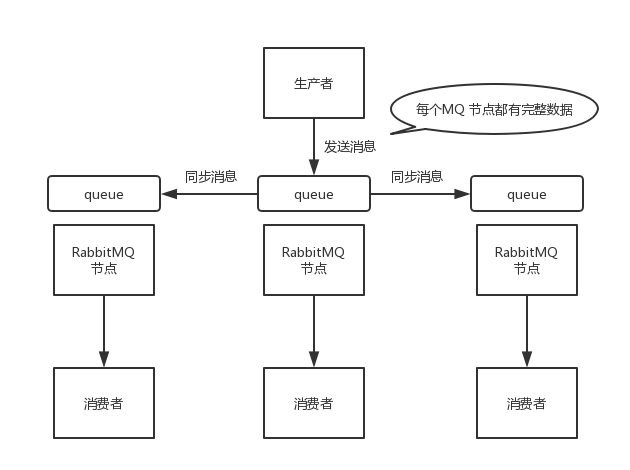
种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个 queue 所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放 queue 的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让 RabbitMQ 落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个 queue 拉取数据。

所以这个事儿就比较尴尬了，这就没有什么所谓的高可用性，这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

#### 镜像集群模式（高可用性）

这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会**存在于多个实例上**，就是说，每个 RabbitMQ 节点都有这个 queue 的一个**完整镜像**，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把**消息同步**到多个实例的 queue 上。



那么**如何开启这个镜像集群模式**呢？其实很简单，RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是**镜像集群模式的策略**，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！第二，这么玩儿，不是分布式的，就**没有扩展性可言**了，如果某个 queue 负载很重，你加机器，新增的机器也包含了这个 queue 的所有数据，并**没有办法线性扩展**你的 queue。你想，如果这个 queue 的数据量很大，大到这个机器上的容量无法容纳了，此时该怎么办呢？

## 5.8消息队列有什么优点和缺点？

**优点：**

系统解耦，消息异步，流量削锋。

**缺点：**

系统可用性降低  
系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？

系统复杂度提高  
硬生生加个 MQ 进来，你怎么[保证消息没有重复消费](https://github.com/liuxc1/advanced-java/blob/master/docs/high-concurrency/how-to-ensure-that-messages-are-not-repeatedly-consumed.md)？怎么[处理消息丢失的情况](https://github.com/liuxc1/advanced-java/blob/master/docs/high-concurrency/how-to-ensure-the-reliable-transmission-of-messages.md)？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

一致性问题  
A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

## 5.9如果让你写一个消息队列，该如何进行架构设计？说一下你的思路。

mq 肯定是很复杂的，面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。从几个方面考虑

1.消息队列的可伸缩性（就是需要的时候快速扩容）

2 .消息队列的高可用性（就是保证一台服务器宕机过后消息队列依然能够正常使用）

1. 消息队列数据的持久性（就是将消息内存数据刷新到磁盘之中保证数据不会丢弃）
2. 保证消息队列消息零丢失问题

# 缓存（redis）

## 6.1项目中缓存是如何使用的？为什么要用缓存？缓存使用不当会造成什么后果？

1.为什么要用缓存？

用缓存的主要有两个目的：1高性能（提高访问效率） 2.高并发

2.缓存使用不当会造成什么后果

\*缓存与数据库双写不一致

\*缓存雪崩、缓存穿透

\*缓存并发竞争

# 分布式

# 数据库

# 9.Sql优化