Java面试手册

目录

5.2、分布式通讯面试整理

5.3、分布式数据库面试整理

此面试文档希望能给正在迷茫时期的Java开发者提供进阶的方向及面试上的储备，我这里有一个Java面试交流微信群，可以讨论技术，内推岗位，以及一些优质资料的分享，可以扫码添加微信邀请进群

正文

一、性能优化专栏

1.1、tomcat性能优化整理

1、你怎样给tomcat调优

1. JVM参数调优： `-Xms<size>` 表示 `JVM` 初始化堆的大小， `-Xmx<size>` 表示 `JVM` 堆的最大值。这两个值的大小一般根据需要进行设

置。当应用程序需要的内存超出堆的最大值时虚拟机就会提示内存溢出，并且导致应用服务崩溃。因此一般建议堆的最大值设置为可用内存的最大值的80%。在 `catalina.bat` 中，设置 `JAVA_OPTS='-Xms256m-Xmx512m'` ，表示初始化内存为256MB，可以使用的最大内存为512MB。

2.禁用DNS查询

当web应用程序向要记录客户端的信息时，它也会记录客户端的IP地址或者通过域名服务器查找机器名转换为IP地址。DNS查询需要占用网络，并且包括可能从很多很远的服务器或者不起作用的服务器上去获取对应的IP的过程，这样会消耗一定的时间。为了消除DNS查询对性能的影响我们可以关闭DNS查询，方式是修改 `server.xml` 文件中的 `enableLookups` 参数值：

```
Tomcat4

<Connector
className="org.apache.coyote.tomcat4.CoyoteConnector"port="80"
minProcessors="5"maxProcessors="75"enableLookups="false"redire
ctPort="8443"
acceptCount="100"debug="0"connectionTimeout="20000"
useURIValidationHack="false"disableUploadTimeout="true"/>

Tomcat5
<Connectorport="80"maxThreads="150"minSpareThreads="25"
maxSpareThreads="75"enableLookups="false"redirectPort="8443"
acceptCount="100"debug="0"connectionTimeout="20000"
disableUploadTimeout="true"/>
```

3.调整线程数

通过应用程序的连接器 `(Con nector)` 进行性能控制的的参数是创建的处理请求的线程数。 `Tomcat` 使用线程池加速响应速度来处理请求。在Java中线程是程序运行时的路径，是在一个程序中与其它控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出CPU最大利用率的高效程序，使空闲时间保持最低，从而接更多的请求。

`Tomcat4` 中可以通过修改 `minProcessors` 和 `maxProcessors` 的值来控制线程数。这些值在安装后就已经设定为默认值并且是足够使用的，但是随着站点的扩容而改大这些值。 `minProcessors` 服务器启动时创建的处理请求的线程数应该足够处理一个小量的负载。也就是说，如果一天内每秒仅发生5次单击事件，并且每个请求任务处理需要1秒钟，那么预先设置线程数为5就足够了。但在你的站点访问量较大时就需要设置更大的线程数，指定为参数 `maxProcessors` 的值。 `maxProcessors` 的值也是有上限的，应防止流量不可控制（或者恶意的服务攻击），从而导致超出了虚拟机使用内存的大小。如果要加大并发连接数，应同时加大这两个参数。 `web server` 允许的最大连接数还受制于操作系统的内核参数设置，通常Windows是2000个左右，Linux是1000个左右。

在 `Tomcat5` 对这些参数进行了调整，请看下面属性：
`maxThreads Tomcat` 使用线程来处理接收的每个请求。这个值表示Tomcat可创建的最大的线程数。

`acceptCount` 指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。

`connnection Timeout` 网络连接超时，单位：毫秒。设置为0表示永不超时，这样设置有隐患的。通常可设置为30000毫秒。

`minSpareThreadsTomcat` 初始化时创建的线程数。

`maxSpareThreads` 一旦创建的线程超过这个值， `Tomcat` 就会关闭不再需要的 `socket` 线程。

最好的方式是多设置几次并且进行测试，观察响应时间和内存使用情况。在不同的机器、操作系统或虚拟机组合的情况下可能会不同，而且并不是所有人的web站点的流量都是一样的，因此没有一刀切的方案来确定线程数的值。

2、如何加大comcat连接数
在 `tomcat` 配置文件 `server.xml` 中的 `<Connector/>`
配置中，和连接数相关的参数有：
`minProcessors` ： 最小空闲连接线程数，用于提高系统处理性能，默认值为10
`maxProcessors` ： 最大连接线程数，即：并发处理的最大请求数，默认值为75

`acceptCount`：允许的最大连接数，应大于等于`maxProcessors`，默认值为100
`enableLookups`：是否反查域名，取值为：true或false。为了提高处理能力，应设置为false
`connectionTimeout`：网络连接超时，单位：毫秒。设置为0表示永不超时，这样设置有隐患的。通常可设置为30000毫秒。
其中和最大连接数相关的参数为maxProcessors和acceptCount。如果要加大并发连接数，应同时加大这两个参数。
`web server`允许的最大连接数还受制于操作系统的内核参数设置，通常Windows是2000个左右，Linux是1000个左右。tomcat5中的配置示例：

```
<Connectorport="8080"
maxThreads="150"minSpareThreads="25"maxSpareThreads="75"
enableLookups="false"redirectPort="8443"acceptCount="100"
debug="0"connectionTimeout="20000"
disableUploadTimeout="true"/>
```

对于其他端口的侦听配置，以此类推。

3、怎样加大tomcat的内存
首先检查程序有没有限入死循环
这个问题主要还是由这个问题`java.lang.OutOfMemoryError:Java heap space`
引起的。第一次出现这样的的问题以后，引发了其他的问题。在网上一查可能是JAVA的堆栈设置太小的原因。
跟据网上的答案大致有这两种解决方法：

1、设置环境变量
解决方法：手动设置`Heap size`
修改`TOMCAT_HOME/bin/catalina.sh`
`setJAVA_OPTS=-Xms32m-Xmx512m`
可以根据自己机器的内存进行更改。

2、`java-Xms32m-Xmx800m className`
就是在执行JAVA类文件时加上这个参数，其中className
是需要执行的确类名。（包括包名）这个解决问题了。而且执行的速度比没有设置的时候快很多。如果在测试的时候可能会用Eclispe这时候就需要

在 Eclipse->run-arguments 中的 VM arguments 中输入 -Xms32m-Xmx800m 这个参数就可以了。
后来在Eclilpse中修改了启动参数，在 VMarguments
加入了 -Xms32m-Xmx800m ，问题解决。

一、 java.lang.OutOfMemoryError:PermGen space
PermGen space 的全称是 Permanent Generation space ,是指内存的永久保存区域,
这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到 PermGen space 中,它和存放类实例(Instance)的Heap区域不同, GC(Garbage Collection) 不会在主程序运行期对 PermGen space 进行清理，所以如果你的应用中有很多CLASS的话,
就很可能出现 PermGen space 错误,这种错误常见在web服务器对JSP进行 preco mpile 的时候。如果你的 WEB APP 下都用了大量的第三方jar,
其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。
解决方法：手动设置MaxPermSize大小修
改 TOMCAT_HOME/bin/catalina.sh
在 “echo"Using CATALINA_BASE:$CATALINA_BASE"” 上面加入以下行： JAVA_OPTS="-server-XX:PermSize=64M-XX:MaxPermSize=128m
建议：将相同的第三方jar文件移置到 tomcat/shared/lib 目录下，这样可以达到减少jar文档重复占用内存的目的。

二、 java.lang.OutOfMemoryError:Java heap space
Heap size 设置
JVM堆的设置是指java程序运行过程中JVM可以调配使用的内存空间的设置.JVM在启动的时候会自动设置 Heap size 的值,
其初始空间 (即-Xms) 是物理内存的1/64，最大空间 (-Xmx) 是物理内存的1/4。可以利用JVM提供的 -Xmn-Xms-Xmx 等选项可进行设置。 Heap size 的大小是 Young Generation 和 TenuredGeneraion
之和。
提示：在JVM中如果98%的时间是用于GC且可用的 Heap size 不足2%的时候将抛出此异常信息。
提示： Heap Size 最大不要超过可用物理内存的80%，一般的要将 -Xms 和 -Xmx 选项设置为相同，而 -Xmn 为1/4的 -Xmx 值。
解决方法：手动设置 Heap size

修改 `TOMCAT_HOME/bin/catalina.sh`
在 `“echo"Using CATALINA_BASE:$CATALINA_BASE"”` 上面加入以下行：
`JAVA_OPTS="-server-Xms800m-Xmx800m-XX:MaxNewSize=256m"`

三、实例，以下给出1G内存环境下 `java jvm` 的参数设置参考：
`JAVA_OPTS="-server-Xms800m-Xmx800m-XX:PermSize=64M-XX:MaxNewSize=256m-XX:MaxPermSize=128m-Djava.awt.headless=true"`
很大的web工程，用tomcat默认分配的内存空间无法启动，如果不是在myeclipse中启动tomcat可以对tomcat
这样设置：
`TOMCAT_HOME/bin/catalina.bat` 中添加这样一句话：
`set JAVA_OPTS=-server-Xms2048m-Xmx4096m-XX:PermSize=512M-XX:MaxPermSize=1024M-Duser.timezone=GMT+08`
或者
`set JAVA_OPTS=-Xmx1024M-Xms512M-XX:MaxPermSize=256m`
如果要在myeclipse中启动，上述的修改就不起作用了，可如下设置：
`Myeclipse->preferences->myeclipse->servers->tomcat->tomcat×.×->JDK` 面板中的 `Optional Java VM arguments` 中添加： `-Xmx1024M-Xms512M-XX:MaxPermSize=256m`
以上是转贴，但本人遇见的问题是：在myeclipse中启动Tomcat时，提示 `"ava.lang.OutOfMemoryError:Java heap space"` ，解决办法就是： `Myeclipse->preferences->myeclipse>servers->tomcat->tomcat×.×->JDK` 面板中的
`Optional Java VM arguments` 中添加： `-Xmx1024M-Xms512M-XX:MaxPermSize=256m`

4、tomcat中如何禁止列目录下的文件
在 `{tomcat_home}/conf/web.xml` 中，把listings参数设置成false即可，如下：

```
<init-param>
<param-name>listings</param-name>
<param-value>false</param-value>
</init-param>
<init-param>
```

```
<param-name>listings</param-name>
<param-value>false</param-value>
</init-param>
```

5、Tomcat有几种部署方式

tomcat中四种部署项目方法

第一种方法:

在 `tomcat` 中的 `conf` 目录中，在 `server.xml` 中的， `<host/>` 节点中添加:

```
<Context path="/hello"
docBase="D:/eclipse3.2.2/forwebtoolsworkspacehello/WebRoot"deb
ug="0"
privileged="true">
</Context>
```

至于 `Context` 节点属性，可详细见相关文档。

第二种方法:

将 `web` 项目文件件拷贝到 `webapps` 目录中。

第三种方法:

很灵活，在 `conf` 目录中，新建 `Catalina` （注意大小写）

\ `localhost` 目录，在该目录中新建一个 `xml` 文件，名字可以随意取，只要和当前文件中的文件名不重复就行了，该 `xml` 文件的内容为:

```
<Context path="/hello"docBase="D:eclipse3.2.2forwebtoolsworksp
aceheloWebRoot"
debug="0"privileged="true">
</Context>
```

第3个方法有个优点，可以定义别名。服务器端运行的项目名称为 `path` ，外部访问的 `URL` 则使用 `XML` 的文件名。这个方法很方便的隐藏了项目的名称，对一些项目名称被固定不能更换，但外部访问时又想换个路径，非常有效。

第2、3还有优点，可以定义一些个性配置，如数据源的配置等。

第四种办法:

可以用 `tomcat` 在线后台管理器，一般 `tomcat` 都打开了，直接上传 `war`

就可以

6、Tomcat的优化经验

`Tomcat` 作为 `web` 服务器，它的处理性能直接关系到用户体验，下面是种常见的优化措施:

• 去掉对 `web.xml` 的监视，把 `jsp` 提前编辑成 `Servlet` 。有富余物理内存的情况，加大 `tomcat` 使用的 `jvm` 的内存。

• 服务器资源
服务器所能提供CPU、内存、硬盘的性能对处理能力有决定性影响。

• 对于高并发情况下会有大量的运算，那么CPU的速度会直接影响到处理速度。

• 内存在大量数据处理的情况下，将会有较大的内存容量需求，可以用 `-Xmx-Xms-XX:MaxPermSize` 等参数对内存不同功能块进行划分。我们之前就遇到过内存分配不足，导致虚拟机一直处于 `full GC` ，从而导致处理能力严重下降。

• 硬盘主要问题就是读写性能，当大量文件进行读写时，磁盘极容易成为性能瓶颈。最好的办法还是利用下面提到的缓存。

• 利用缓存和压缩
对于静态页面最好是能够缓存起来，这样就不必每次从磁盘上读。这里我们采用了Nginx作为缓存服务器，将图片、css、js文件都进行了缓存，有效的减少了后端tomcat的访问。另外，为了能加快网络传输速度，开启gzip压缩也是必不可少的。但考虑到tomcat已经需要处理很多东西了，所以把这个压缩的工作就交给前端的Nginx来完成。

除了文本可以用gzip压缩，其实很多图片也可以用图像处理工具预先进行压缩，找到一个平衡点可以让画质损失很小而文件可以减小很多。曾经我就见过一个图片从300多kb压缩到几十kb，自己几乎看不出来区别。

• 采用集群
单个服务器性能总是有限的，最好的办法自然是实现横向扩展，那么组建tomcat集群是有效提升性能的手段。我们还是采用了Nginx来作为请求分流的服务器，后端多个tomcat共享session来协同工作。可以参考之前写的《利用nginx+tomcat+memcached组建web服务器负载均衡》。

• 优化tomcat参数

这里以tomcat7的参数配置为例，需要修改conf/server.xml文件，主要是优化连接配置，关闭客户端dns查询。

```
<Connector port="8080"
protocol="org.apache.coyote.http11.Http11NioProtocol"
connectionTimeout="20000"
redirectPort="8443"
maxThreads="500"
minSpareThreads="20"
acceptCount="100"
disableUploadTimeout="true"
enableLookups="false"
URIEncoding="UTF-8"/>
```

1.2、JVM性能优化专题

1、Java类加载过程

Java 类加载需要经历一下7 个过程：

1 . 加载

加载是类加载的第一个过程，在这个阶段，将完成一下三件事情：

- 通过一个类的全限定名获取该类的二进制流。
- 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
- 在内存中生成该类的 `Class` 对象，作为该类的数据访问入口。

2 . 验证

验证的目的是为了确保 `Class` 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

- 文件格式验证：验证字节流是否符合 `Class` 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
- 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
- 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方

法中的类型转换是否正确，跳转指令是否正确等。
• 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

3 . 准备
准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java 堆中。

```
public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。
```

4 . 解析
该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

5 . 初始化
初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中的定义的java程序代码。

6.使用

7.卸载

2、java内存分配
• 寄存器：我们无法控制。
• 静态域：`static` 定义的静态成员。
• 常量池：编译时被确定并保存在.`class` 文件中的 `(final)` 常量值和一些文本修饰的符号引用（类和接口的全限定名，字段的名称和描述符，方法和名称和描述符）。
• 非`RAM` 存储：硬盘等永久存储空间。
• 堆内存：new 创建的对象和数组，由Java 虚拟机自动垃圾回收器管理，存取速度慢。
• 栈内存：基本类型的变量和对象的引用变量（堆内存空间的访问地址），速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

1.Java 堆的结构是什么样子的？什么是堆中的永久代 `(Perm Gen space)` ？

`JVM` 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 `JVM` 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。

3、描述一下JVM加载Class文件的原理机制?

Java 语言是一种具有动态性的解释型语言，类（Class）只有被加载到JVM 后才能运行。当运行指定程序时，JVM 会将编译生成的 `.class` 文件按照需求和一定的规则加载到内存中，并组织成为一个完整的Java 应用程序。这个加载过程是由类加载器完成，具体来说，就是由ClassLoader 和它的子类来实现的。类加载器本身也是一个类，其实质是把类文件从硬盘读取到内存中。

类的加载方式分为隐式加载和显示加载。隐式加载指的是程序在使用new 等方式创建对象时，会隐式地调用类的加载器把对应的类加载到JVM 中。显示加载指的是通过直接调用 `class.forName()` 方法来把所需的类加载到JVM 中。

任何一个工程项目都是由许多类组成的，当程序启动时，只把需要的类加载到JVM 中，其他类只有被使用到的时候才会被加载，采用这种方法一方面可以加快加载速度，另一方面可以节约程序运行时对内存的开销。此外，在Java 语言中，每个类或接口都对应一个 `.class` 文件，这些文件可以被看成是一个个可以被动态加载的单元，因此当只有部分类被修改时，只需要重新编译变化的类即可，而不需要重新编译所有文件，因此加快了编译速度。

在Java 语言中，类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类（例如基类）完全加载到JVM 中，至于其他类，则在需要的时候才加载。

类加载的主要步骤：

- 装载。根据查找路径找到相应的class 文件，然后导入。
- 链接。链接又可分为3 个小步：
- 检查，检查待加载的class 文件的正确性。

• 准备，给类中的静态变量分配存储空间。
• 解析，将符号引用转换为直接引用（这一步可选）
• 初始化。对静态变量和静态代码块执行初始化工作。

4、GC 是什么? 为什么要有 GC?

GC 是垃圾收集的意思(GabageCollection)，内存处理是编程 人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或 系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象 是否超过作用域从而达到自动回收内存的目的，Java 语言没有提 供释放已分配内存的显示操作方法。

5、简述 Java 垃圾回收机制。

在 Java 中，程序员是不需要显示的去释放一个对象的内存的，而 是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低 优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当 前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象， 并将它们添加到要回收的集合中，进行回收。

6、 如何判断一个对象是否存活?(或者 GC 对象的判定方法)

判断一个对象是否存活有两种方法:

1 . 引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一 个地方引用这个对象时，就将计数器加一，引用失效时，计数器就 减一。当一个对象的引用计数器为零时，说明此对象没有被引用， 也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对 象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A、B 对 象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流 的虚拟机都没有采用这种算法。

2 . 可达性算法(引用链法)

该算法的思想是:从一个被称为 GC Roots 的对象开始向下搜索， 如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对 象不可用。

在 Java 中可以作为 GC Roots 的对象有以下几种:

• 虚拟机栈中引用的对象
• 方法区类静态属性引用的对象 • 方法区常量池引用的对象
• 本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条 件时，一个对象比不一定会被回收。当一个对象不可达 GC Root 时，这个对象并不会立马被回收，而是出于一个死缓的阶段，若要 被真正的回收需要经历两次标记.

如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就 会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 `finalize()` 方法。当对象没有覆盖 `finalize()` 方法或者已被虚拟机调用过，那么就认为是没必要的。 如果该对象有必要执行 `finalize()` 方法，那么这个对象将会放在一个称为 F-Queue 的对 队列中，虚拟机会触发一个 `Finalize()` 线程去执行，此线程是低 优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize() 执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行第二次被标记，这时，该对象将被移除” 即将回收” 集合，等待回收。

7、垃圾回收的优点和原理。并考虑 2 种回收机制。

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 C++ 程序员最头疼的内存管理的问题迎刃而解，它使得 Java 程序员在 编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制， Java 中的对象不再有“作用域”的概念，只有对象的引用才有"作用域"。垃圾回收可以有效的防止内存泄露，有效的使用可以使 用的内存。垃圾回收器通常是作为一个单独的低级别的线程运行， 不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的 对象进行清楚和回收，程序员不能实时的调用垃圾回收器对某个对 象或所有对象进行垃圾回收。

回收机制有分代复制垃圾回收和标记垃圾回收，增量垃圾回收。

8、垃圾回收器的基本原理是什么?垃圾回收器可以马上回收内存吗? 有什么办法主动通知虚拟机进行垃圾回收?

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象 的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和 管理堆(heap)中的所有对象。通过这种方式确定哪些对象是” 可达的”，哪些对象是”不可达的”。当 GC 确定一些对象为“不 可达”时，GC 就有责任回收这些内存空间。可以。程序员可以手动执行 `System.gc()`，通知 GC 运行，但是 Java 语言规范并不 保证 GC 一定会执行。

9、Java 中会存在内存泄漏吗，请简单描述。

所谓内存泄露就是指一个不再被程序使用的对象或变量一直被占 据在内存中。Java 中有垃圾回收机制，它可以保证一对象不再被 引用的时候，即对象变成了孤儿的时候，对象将自动被垃圾回收器 从内存中清除掉。由于 Java 使用有向图的方式进行垃圾回收管理， 可以消除引用循环的问题，例如有两个对象，相互引用，只要它们 和根进程不可达的，那么 GC 也是可以回收它们的，例如下面的 代码可以看到这种情况的内存回收:

```
import java.io.IOException; public class GarbageTest {
   /**
    * @param args
    * @throws IOException
    */
public static void main(String[] args) throws IOException {
       // TODO Auto-generated method stub
try { gcTest();
} catch (IOException e) {
// TODO Auto-generated catch block
     e.printStackTrace();
   }
System.out.println("has exited gcTest!"); System.in.read();
System.in.read(); System.out.println("out begin gc!"); for(int
 i=0;i<100;i++)
   {
       System.gc();
System.in.read();
System.in.read(); }
}
private static void gcTest() throws IOException {
System.in.read();
System.in.read();
Person p1 = new Person(); System.in.read();
System.in.read();
Person p2 = new Person();
p1.setMate(p2);
p2.setMate(p1);
```

```
System.out.println("before exit gctest!"); System.in.read();
System.in.read();
System.gc(); System.out.println("exit gctest!");
}
private static class Person {
byte[] data = new byte[20000000]; Person mate = null;
public void setMate(Person other) {
          mate = other;
       }
     }
  }
```

Java 中的内存泄露的情况:长生命周期的对象持有短生命周期对 象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需 要，但是因为长生命周期对象持有它的引用而导致不能被回收，这 就是 Java 中内存泄露的发生场景，通俗地说，就是程序员可能创 建了一个对象，以后一直不再使用这个对象，这个对象却一直被引 用，即这个对象无用但是却无法被垃圾回收器回收的，这就是 java 中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对 象放在缓存中 (例如放在一个全局 map 对象中)，然后一直不再 使用它，这个对象一直被缓存引用，但却不再被使用。

检查 Java 中的内存泄露，一定要让程序将各种分支情况都完整执 行到程序结束，然后看某个对象是否被使用过，如果没有，则才能 判定这个对象属于内存泄露。

如果一个外部类的实例对象的方法返回了一个内部类的实例对象， 这个内部类对象被长期引用了，即使那个外部类实例对象不再被使 用，但由于内部类持久外部类的实例对象，这个外部类对象将不会 被垃圾回收，这也会造成内存泄露。

下面内容来自于网上(主要特点就是清空堆栈中的某个元素，并不 是彻底把它从数组中拿掉，而是把存储的总数减少，本人写得可以 比这个好，在拿掉某个元素时，顺便也让它从数组中消失，将那个 元素所在的位置的值设置为 null 即可):

我实在想不到比那个堆栈更经典的例子了，以致于我还要引用别人 的例子，下面的例子不是我想到的，是书上看到的，当然如果没有 在书上看到，可能过一段时间我自己也想的到，可是那时我说是我 自己想到的也没

有人相信的。

```
public class Stack {
private Object[] elements=new Object[10]; private int size = 0
;
public void push(Object e){
       ensureCapacity();
       elements[size++] = e;
       }
public Object pop(){
if( size == 0) throw new EmptyStackException(); return element
s[--size];
}
private void ensureCapacity(){ if(elements.length == size){
Object[] oldElements = elements;
elements = new Object[2 * elements.length+1]; System.arraycopy
(oldElements,0, elements, 0,
size);
       }
    }
 }
```

上面的原理应该很简单，假如堆栈加了 10 个元素，然后全部弹 出来，虽然堆栈是空的，没有我们要的东西，但是这是个对象是无 法回收的，这个才符合了内存泄露的两个条件:无用，无法回收。 但是就是存在这样的东西也不一定会导致什么样的后果，如果这个 堆栈用的比较少，也就浪费了几个 K 内存而已，反正我们的内存都 上 G 了，哪里会有什么影响，再说这个东西很快就会被回收的， 有什么关系。下面看两个例子。

```
public class Bad{
public static Stack s=Stack(); static{
s.push(new Object());
s.pop(); //这里有一个对象发生内存泄露
s.push(new Object()); //上面的对象可以被回收了，等于是自 愈了
        }
   }
```

因为是 static，就一直存在到程序退出，但是我们也可以看到它有 自愈功能，就是说如果你的 Stack 最多有 100 个对象，那么最 多也就只有 100 个对象无法被回收其实这个应该很容易理解， Stack 内部持有 100 个引用，最坏的情况就是他们都是无用的， 因为我们一旦放新的进取，以前的引用自然消失!

内存泄露的另外一种情况:当一个对象被存储进 HashSet 集合中 以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否 则，对象修改后的哈希值与最初存储进 HashSet 集合中时的哈希 值就不同了，在这种情况下，即使在 contains 方法使用该对象的 当前引用作为的参数去 HashSet 集合中检索对象，也将返回找不 到对象的结果，这也会导致无法从 HashSet 集合中单独删除当前 对象，造成内存泄露。

10、深拷贝和浅拷贝。

简单来讲就是复制、克隆。

```
Person p=new Person(“张三”);
```

浅拷贝就是对对象中的数据成员进行简单赋值，如果存在动态成员或者指针就会报错。

深拷贝就是对对象中存在的动态成员或指针重新开辟内存空间。

11、System.gc() 和 Runtime.gc() 会做什么事情?

这两个方法用来提示 JVM 要进行垃圾回收。但是，立即开始还是 延迟进行垃圾回收是取决于 JVM 的。

12、finalize() 方法什么时候被调用?析构函数 (finalization) 的 目的是什么?

垃圾回收器(garbage colector)决定回收某对象时，就会运行该 对象的 finalize() 方法 但是在 Java 中很不幸，如果内存总是充 足的，那么垃圾回收可能永远不会进行，也就是说 filalize() 可能 永远不被执行，显然指望它做收尾工作是靠不住的。 那么 finalize() 究竟是做什么的呢? 它最主要的用途是回收特殊渠道 申请的内存。Java 程序有垃圾回收器，所以一般情况下内存问题 不用程序员操心。但有一种 JNI(Java Native Interface)调用 non-Java 程序(C 或 C++)， finalize() 的工作就是回收这部 分的内存。

13、如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占 用的内存?

不会，在下一个垃圾回收周期中，这个对象将是可被回收的。

14、什么是分布式垃圾回收(DGC)?它是如何工作的?
DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。 因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难 的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

15串行(serial)收集器和吞吐量(throughput)收集器的区别 是什么?
吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模 和大规模数据的应用程序。 而串行收集器对大多数的小应用(在 现代处理器上需要大概 100M 左右的内存)就足够了。

16、在 Java 中，对象什么时候可以被垃圾回收?
当对象对当前使用这个对象的应用程序变得不可触及的时候，这个 对象就可以被回收了。

17、简述 Java 内存分配与回收策率以及 Minor GC 和 Major GC。
- 对象优先在堆的 Eden 区分配
- 大对象直接进入老年代
- 长期存活的对象将直接进入老年代

当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC。Minor GC 通常发生在新生代的 Eden 区，在这个区 的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快; Full GC/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC，但是通过配置，可以在 Full GC 之 前进行一次 Minor GC 这样可以加快老年代的回收速度。

18、JVM 的永久代中会发生垃圾回收么?
垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值， 会触发完全垃圾回收(Full GC)。
注: Java 8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区。

19、Java 中垃圾收集的方法有哪些?
标记 - 清除:这是垃圾收集算法中最基础的，根据名字就可以知 道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种 方法很简单，但是会有两个主要问题:
1. 效率不高，标记和清除的效率都很低;
2. 会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作。

复制算法:为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。

于是将该算法进行了改进，内存区域不再是按照 1:1 去划分，而 是将内存划分为 8:1:1 三部分，较大那份内存交 Eden 区，其余 是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区， 若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这 些对象通过分配担保机制复制到老年代中。(java 堆又分为新生代和老年代)

标记 - 整理:该算法主要是为了解决标记 - 清除，产生大量内存 碎片的问题;当对象存活率较高时，也解决了复制算法的效率问题。 它的不同之处就是在清除对象的时候现将可回收对象移动到一端， 然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

分代收集:现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。 老年代里的对象存活率较高，没有额外的空间进行分配担保。

20、什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。主要有一下四种类加载器:

- 启动类加载器(BootstrapClassLoader)用来加载Java核 心类库，无法被 Java 程序直接引用。
- 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类 加载器在此目录里面查找并加载 Java 类。
- 系统类加载器(system class loader):它根据 Java 应用 的类路径(CLASSPATH)来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。
- 用户自定义类加载器，通过继承 java.lang.ClassLoader 类 的方式实现。

21、类加载器双亲委派模型机制?

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派

给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

3、Mysql性能优化整理

二、微服务架构面试专栏

1、SpringCloud面试整理

1、什么是 Spring Cloud?

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。 Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

2、使用 Spring Cloud 有什么优势?

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

- 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- 冗余-分布式系统中的冗余问题。
- 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央 处理单元，或磁盘驱动器的分布。
- 性能-问题 由于各种运营开销导致的性能问题。
- 部署复杂性-Devops 技能的要求。

3、服务注册和发现是什么意思?Spring Cloud 如何实现?

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

4、负载平衡的意义什么?

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算

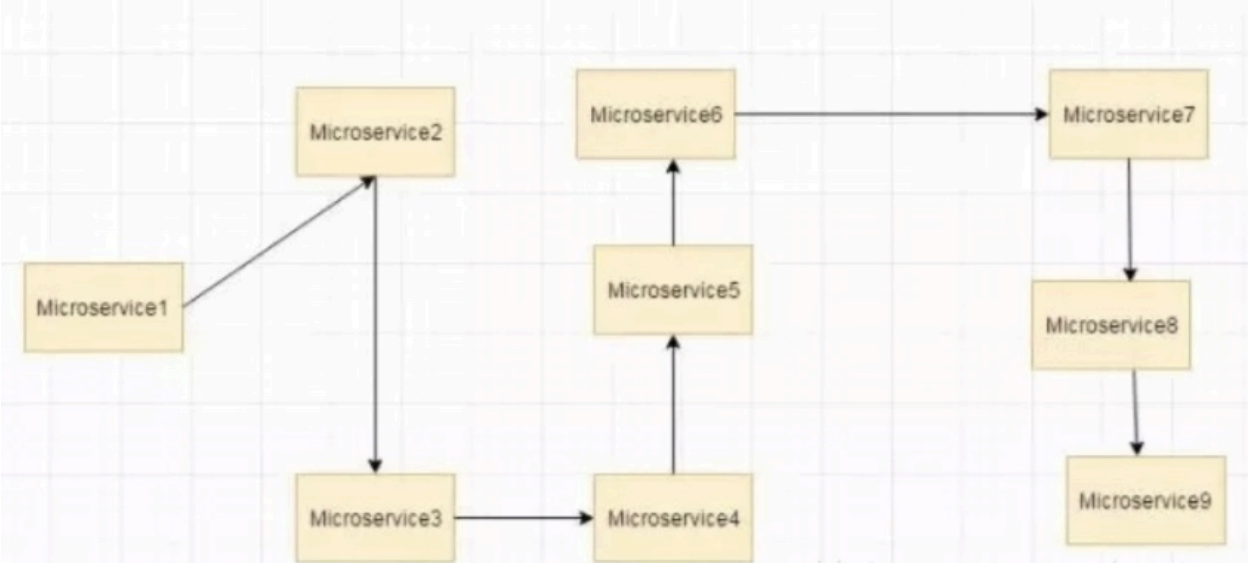
资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源
的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

5、什么是 Hystrix?它如何实现容错?

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

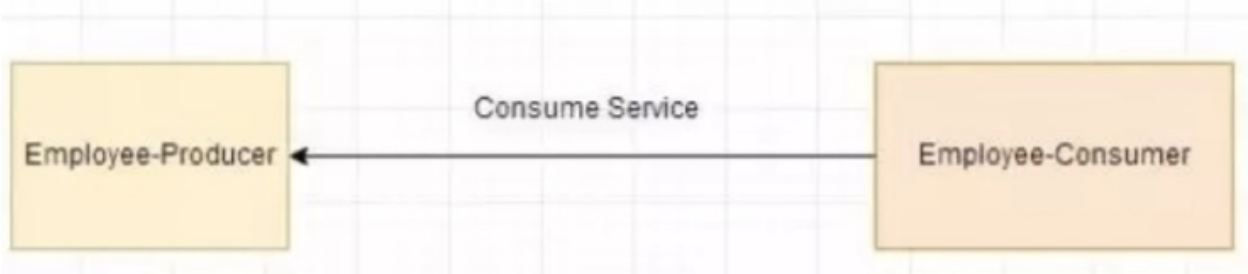
思考以下微服务



假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方， 我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

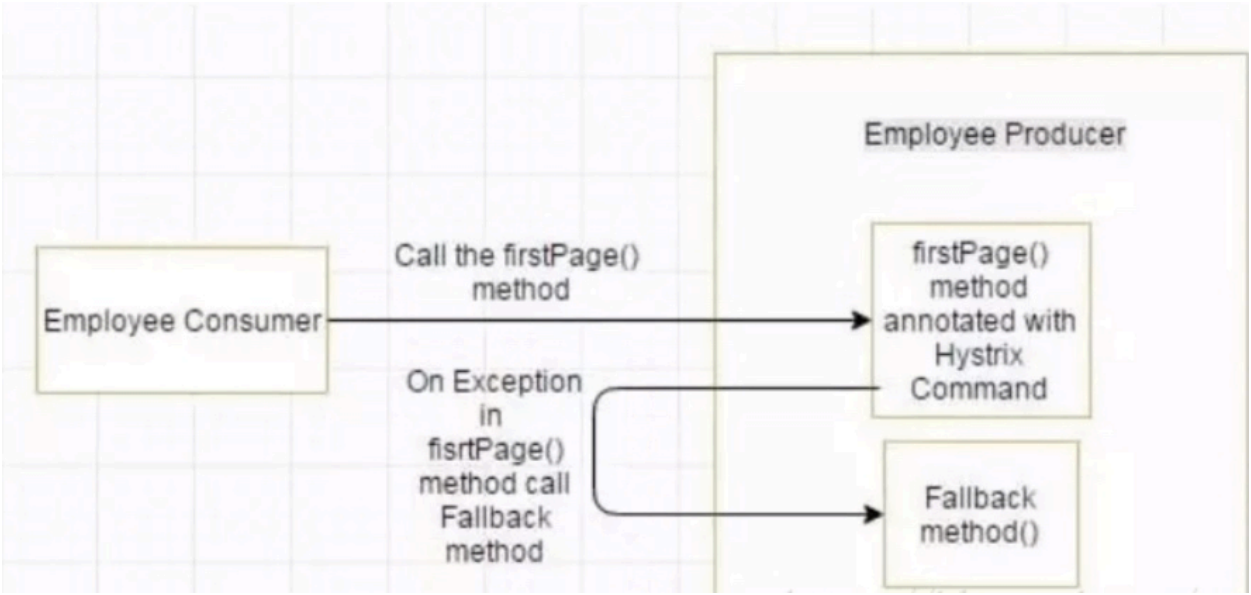
简化图如下所示



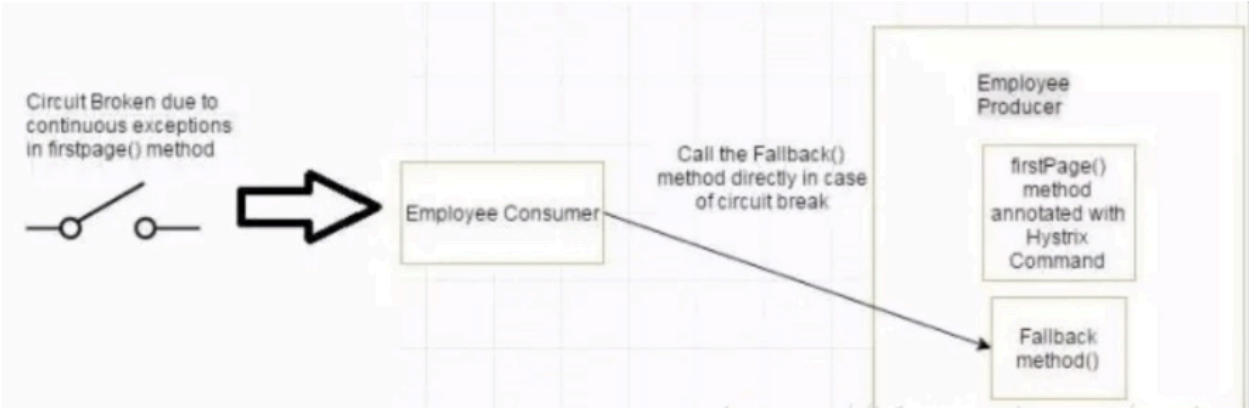
现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

6、什么是 Hystrix 断路器?我们需要它吗?

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个 回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firtsPage 方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法 留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。



7、什么是 Netflix Feign?它的优点是什么?

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不

考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。
但是我们必须编写大量代码才能执行以下步骤

- 使用功能区进行负载平衡。
- 获取服务实例，然后获取基本 URL。
- 利用 REST 模板来使用服务。前面的代码如下

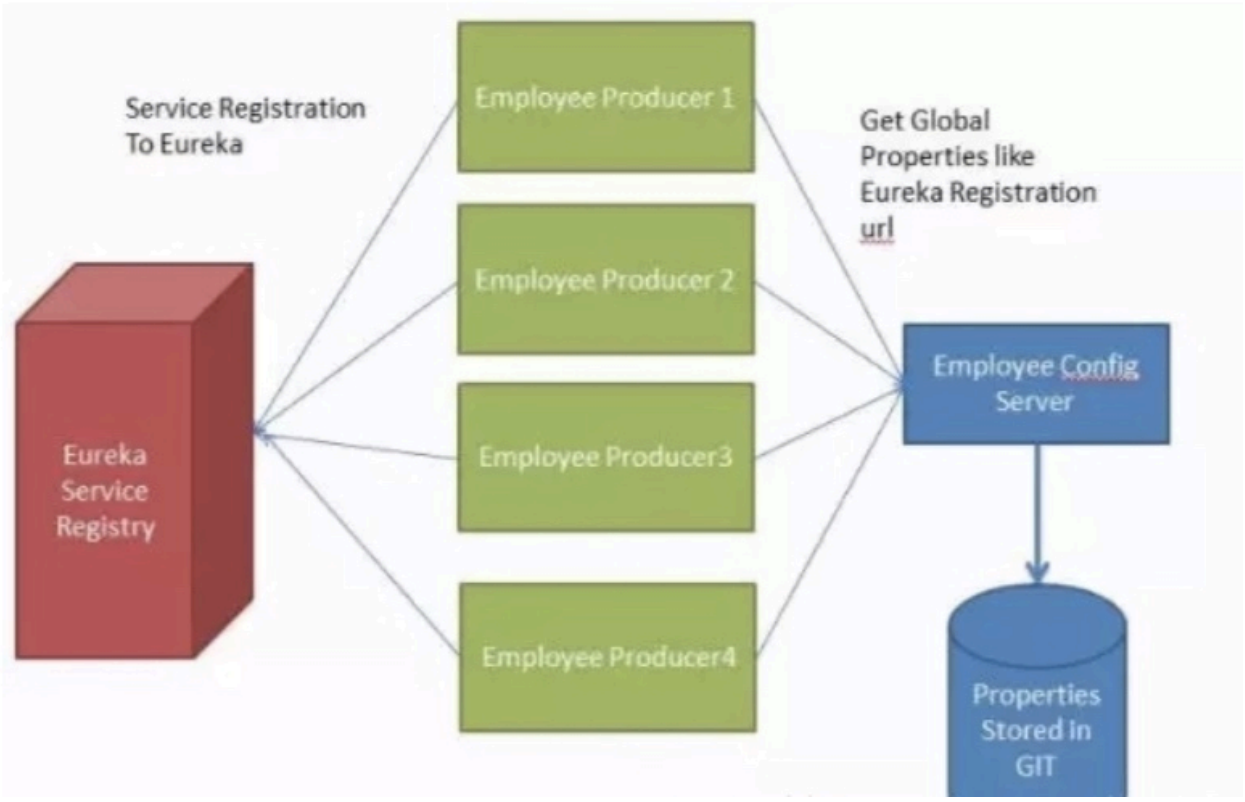
```
1. @Controller
2. public class ConsumerControllerClient {
3.
4. @Autowired
5. private LoadBalancerClient loadBalancer;
6.
7. public void getEmployee() throws RestClientException, IOExc
eption {
8.
9. ServiceInstance serviceInstance=loadBalancer.choose("employ
ee-producer"); 10.
11. System.out.println(serviceInstance.getUri());
12.
13. String baseUrl=serviceInstance.getUri().toString();
14.
15. baseUrl=baseUrl+"/employee";
16.
17. RestTemplate restTemplate = new RestTemplate();
18. ResponseEntity<String> response=null;
19. try{
20. response=restTemplate.exchange(baseUrl,
21. HttpMethod.GET, getHeaders(),String.class);
22. }catch (Exception ex)
23. {
24. System.out.println(ex);
25. }
26. System.out.println(response.getBody());
27. }
```

之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

8、什么是 Spring Cloud Bus?我们需要它吗?

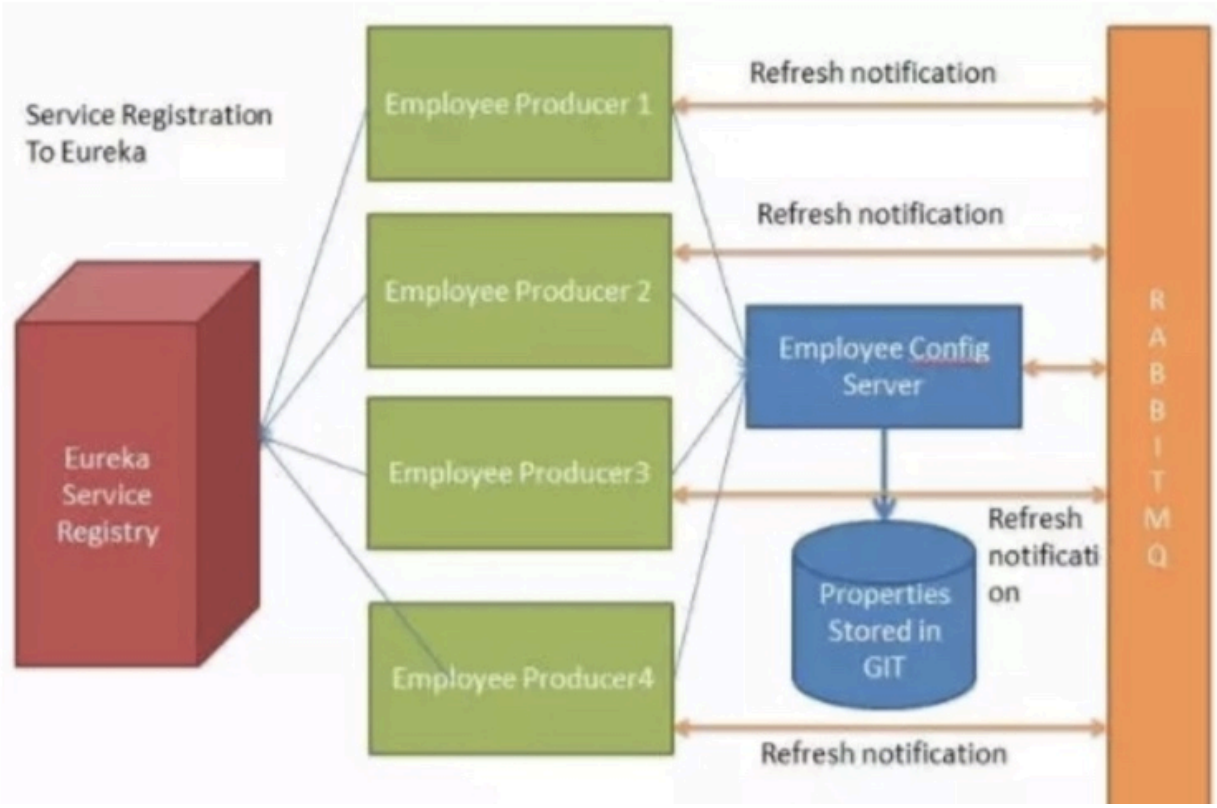
考虑以下情况:我们有多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 http:// localhost:8080 / refresh。同样对于 Employee Producer2 http:// localhost:8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别 有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听 此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

2、SpringBoot面试整理

1、什么是 Spring Boot?

多年来，随着新功能的增加，spring 变得越来越复杂。只需访问 https://spring.io/projects 页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目，我们必须添加构建路径或添加 Maven 依赖关系，配置应用程序服务器，添加 spring 配置。因此，开始一个新的 spring 项目需要很多努力，因为我们现 在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。Spring Boot 已经建立在现有 spring 框架之上。使用spring 启动，我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot可以帮助我们以最少的工作量，更加健壮地使用现有的 Spring 功能。

2、Spring Boot 有哪些优点?

Spring Boot 的优点有:
减少开发，测试时间和努力。
使用 JavaConfig 有助于避免使用 XML。
避免大量的 Maven 导入和各种版本冲突。
提供意见发展方法。
通过提供默认值快速开始开发。
没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东西。
需要更少的配置 因为没有 web.xml 文件。只需添加用@ Configuration 注释的类，然后添加用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将 @Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。 基于环境的配置使用这些属性，您可以将您正在使用的环境传递到应用程序:- Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后，Spring 将在(application{environment} .properties)中加载后续的应用程序属性文件。

3、什么是 JavaConfig?

Spring JavaConfig是Spring社区的产品，它提供了配置Spring IoC容器的纯Java方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于:
面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。
减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配 置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置 容器是可行的，但实际上很多人认为将 JavaConfig 与 XML 混合匹配是理想的。 类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或 基于字符串的查找。

4、如何重新加载 Spring Boot 上的更改，而无需重新启动服务器?

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat 将重新启动。Spring Boot 有一个开发工具(DevTools)模块，它有助于提高开发人员的生 产力。Java 开发人员面临的一个主要挑

战是将文件更改自动部署到服务器并自动重启服务 器。开发人员可以重新加载Spring Boot上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

org.springframework.boot spring-boot-devtools true

5、Spring Boot 中的监视器是什么?

Spring boot actuator是spring启动框架中的重要功能之一。Spring boot监视器可帮助您访 问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和 监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器 模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

6、如何在 Spring Boot 中禁用 Actuator 端点安全性?

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。我们可以使用

management.security.enabled = false

来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

7、如何在自定义端口上运行 Spring Boot 应用程序?

为了在自定义端口上运行 Spring Boot 应用程序，您可以在 application.properties 中指定端口。

server.port = 8090

8、什么是 YAML?

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

9、如何实现 Spring Boot 应用程序的安全性?

为了实现Spring Boot的安全性，我们使用 spring-boot-starter-security依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展 WebSecurityConfigurerAdapter 并覆 盖其方法。

10、如何集成 Spring Boot 和 ActiveMQ?

对于集成 Spring Boot 和 ActiveMQ，我们使用
spring-boot-starter-activemq
依赖关系。 它只需要很少的配置，并且不需要样板代码。

11、如何使用 Spring Boot 实现分页和排序?

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的 org.springframework.data.domain.Pageable
传递给存储库方法。

12、什么是 Swagger?你用 Spring Boot 实现了它吗?

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻 辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

13、什么是 Spring Profiles?

Spring Profiles 允许用户根据配置文件(dev，test，prod 等)来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION 中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

14、什么是 Spring Batch?

Spring Boot Batch提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟 踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

15、什么是 FreeMarker 模板?

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用 freemarker 可以将这些结合起来，给出最终的输出页面。

16、如何使用 Spring Boot 实现异常处理?

Spring提供了一种使用ControllerAdvice处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

17、您使用了哪些 starter maven 依赖项?

使用了下面的一些依赖项

```
spring-boot-starter-activemq spring-boot-starter-security
```

spring-boot-starter-web 这有助于增加更少的依赖关系，并减少版本的冲突。

18、什么是 CSRF 攻击?

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的 Web 应用 程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

19、什么是 WebSockets?

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。

WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

单个TCP连接 -初始连接使用HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

Light -与 http 相比，WebSocket 消息数据交换要轻得多。

20、什么是 AOP?

在软件开发过程中，跨越应用程序多个点的功能称为交叉问题。这些交叉问题与应用程序的主要业务逻辑不同。因此，将这些横切关注与业务逻辑分开是面向方面编程(AOP)的 地方。

21、什么是 Apache Kafka?

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅 消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离 线和在线消息消费。

22、我们如何监视所有 Spring Boot 微服务?

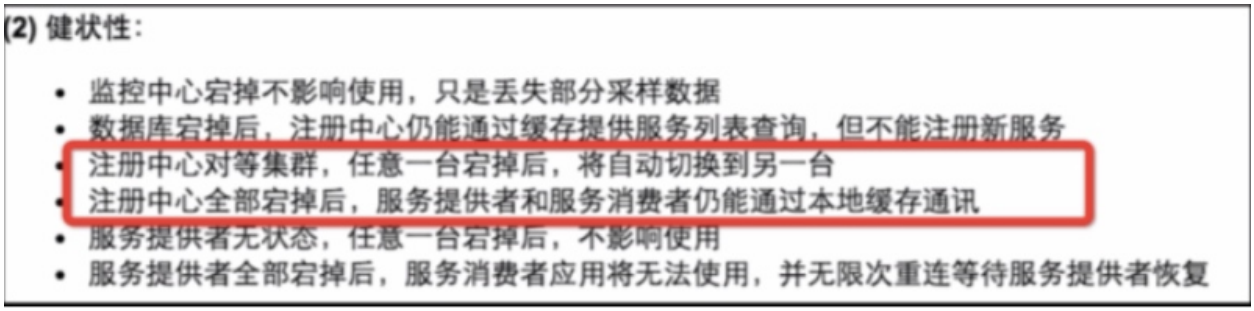
Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息(如它们是否已启动)以及它们的组件(如数据库等)是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及

50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。

3、Dubbo面试整理

1、Dubbo 中 zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么?

可以通信的，启动 dubbo 时，消费者会从 zk 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用; 注册中心对等集群，任意一台宕机后，将会切换到另一台;注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台宕机后，不影响使用;服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复; 挂掉是不要紧的，但前提是你没有增加新的服务，如果你要调用新的服务，则是不能办到的。


(2) 健状性:

- 监控中心宕掉不影响使用，只是丢失部分采样数据
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
- 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
- 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

2、dubbo 服务负载均衡策略?

l Random LoadBalance

随机，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比 较均匀，有利于动态调整提供者权重。(权重可以在 dubbo 管控台配置)

l RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。存在慢的提供者累积请求问题，比如:第二台机器很慢，但没挂，当请求调

到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

l LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的

调用前后计数差会越大。

l ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会

引起剧烈变动。缺省只对第一个参数 Hash，如果要修改，请配置

```
<dubbo:parameter key="hash.arguments" value="0,1" />
```

缺省用 160 份虚拟节点，如果要修改，请配置

```
<dubbo:parameter key="hash.nodes" value="320" />
```

3、 Dubbo 在安全机制方面是如何解决的

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

4、dubbo 连接注册中心和直连的区别

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，

l Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

[AppleScript] 纯文本查看复制代码 ？

服务注册中心，动态的注册和发现服务，使服务的位置透明，并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover， 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给 消费者。服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外，注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

1. dubbo 服务集群配置(集群容错模式)

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。

可以自行扩展集群容错策略 l Failover Cluster(默认)

失败自动切换，当出现失败，重试其它服务器。(缺省)通常用于读操作，但重试会带来更长延迟。可通过 retries="2"来设置重试次数(不含第一次)。

```
<dubbo:service retries="2" cluster="failover"/>
或:
<dubbo:reference retries="2" cluster="failover"/> cluster="fai
lover"可以不用写,因为默认就是 failover
```

l Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，

比如新增记录。

```
dubbo:service cluster="failfast" />
或:
<dubbo:reference cluster="failfast" />
cluster="failfast"和 把 cluster="failover"、retries="0"是一样的效
果,retries="0"就是不重试
```

l Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

```
<dubbo:service cluster="failsafe" />
或:
<dubbo:reference cluster="failsafe" />
```

l Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

```
<dubbo:service cluster="failback" />
或:
<dubbo:reference cluster="failback" />
```

l Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2"来设置最大并行数。

```
<dubbo:service cluster="forking" forks="2"/>
或:
<dubbo:reference cluster="forking" forks="2"/>
```

l 配置

```
服务端服务级别
    <dubbo:service interface="..." loadbalance="roundrobin" />
  客户端服务级别
    <dubbo:reference interface="..." loadbalance="roundrobin"
/>   服务端方法级别   <dubbo:service interface="..."> <dubbo:meth
od name="..." loadbalance 客户端方法级别  <dubbo:reference interf
ace="..."> <dubbo:method name="..." loadbalance="
```

1. dubbo 通信协议 dubbo 协议为什么要消费者比提供者个数多: 因 dubbo 协议采用单一长连接，假设网络为千兆网卡(1024Mbit=128MByte)，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。
2. dubbo 通信协议 dubbo 协议为什么不能传大包: 因 dubbo 协议采用单一长连接， 如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡(1024Mbit=128MByte)，每条连接最大 7MByte(不同的环境可能不一样，供参考)， 单个服务提供者的 TPS(每秒处理事务数)最大为:128MByte / 500KByte = 262。 单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为:7MByte / 500KByte = 14。 如果能接受，可以考虑使用，否则网络将成为瓶颈。
3. dubbo 通信协议 dubbo 协议为什么采用异步单一长连接: 因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务， 比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者， 长连接，减少连接握手验证等， 并使用异步 IO，复用线程池，防止 C10K 问题。
4. dubbo 通信协议 dubbo 协议适用范围和适用场景适用范围:传入传出参数数据包较小(建议小于 100K)，消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。 适用场景:常规远程服务方法调用 dubbo 协议补充: 连接个数:单连接连接方式:长连接传输协议:TCP 传输方式:NIO 异步传输序列化:Hessian 二进制序列化
5. RMI 协议 RMI 协议采用 JDK 标准的 java.rmi.*实现，采用阻塞式短连接

和 JDK 标准序列 化方式，Java 标准的远程调用协议。 连接个数:多连接连接方式:短连接 传输协议:TCP 传输方式:同步传输 序列化:Java 标准二进制序列化 适用范围:传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。 适用场景:常规远程服务方法调用，与原生 RMI 服务互操作

6. Hessian 协议 Hessian 协议用于集成 Hessian 的服务，Hessian 底层采用 Http 通讯，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现 基于 Hessian 的远程调用协议。 连接个数:多连接 连接方式:短连接 传输协议:HTTP 传输方式:同步传输序列化:Hessian 二进制序列化 适用范围:传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。 适用场景:页面传输，文件传输，或与原生 hessian 服务互操作
7. http 采用 Spring 的 HttpInvoker 实现 基于 http 表单的远程调用协议。连接个数:多连接 连接方式:短连接 传输协议:HTTP 传输方式:同步传输序列化:表单序列化(JSON) 适用范围:传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或 URL 传入参数，暂不支持传文件。适用场景:需同时给应用程序和浏览器 JS 使用的服务。
8. Webservice 基于 CXF 的 frontend-simple 和 transports-http 实现 基于 WebService 的远程调用协议。 连接个数:多连接 连接方式:短连接 传输协议:HTTP 传输方式:同步传输 序列化:SOAP 文本序列化适用场景:系统集成，跨语言调用。
9. Thrif Thrift 是 Facebook 捐给 Apache 的一个 RPC 框架，当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信 息，比如 service name，magic number 等

三、并发编程高级面试专栏

1、Synchronized用过吗，其原理是什么?

这是一道Java面试中几乎百分百会问到的问题，因为没有任何写过并发程序的开发者会没听说或者没接触过Synchronized。Synchronized是由JVM实现的一种实现互斥同步的一种方式，如果你查看被Synchronized修饰过的程序块编译后的字节码，会发现，被Synchronized修饰过的程序块，在编译前后被编译器生成了monitorenter和monitorexit两个字节码指令。这两个指令是什么意思呢？在虚拟机执行到monitorenter指令时，首先要尝试

获取对象的锁：如果这个对象没有锁定，或者当前线程已经拥有了这个对象的锁，把锁的计数器+ 1；当执行monitorexit指令时将锁计数器-1；当计数器为0时，锁就被释放了。如果获取对象失败了，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。Java中Synchronize通过在对象头设置标记，达到了获取锁和释放锁的目的。

2、你刚才提到获取对象的锁，这个 “锁 ”到底是什么？如何确定对象的锁?

“锁 ”的本质其实是monitorenter和monitorexit字节码指令的一个Reference类型的参数，即要锁定和解锁的对象。我们知道，使用Synchronized 可以修饰不同的对象，因此，对应的对象锁可以这么确定。

1. 如果 Synchronized 明确指定了锁对象，比如 Synchronized(变量名)、Synchronized(this) 等，说明加解锁对象为该对象。

2. 如果没有明确指定:

若 Synchronized 修饰的方法为非静态方法，表示此方法对应的对象为 锁对象;

若 Synchronized 修饰的方法为静态方法，则表示此方法对应的类对象 为锁对象。

注意，当一个对象被锁住时，对象里面所有用 Synchronized 修饰的 方法都将产生堵塞，而对象里非 Synchronized 修饰的方法可正常被 调用，不受锁影响。

3、什么是可重入性，为什么说 Synchronized 是可重入锁?

可重入性是锁的一个基本要求，是为了解决自己锁死自己的情况。

比如下面的伪代码，一个类中的同步方法调用另一个同步方法，假如 Synchronized 不支持重入，进入 method2 方法时当前线程获得锁，method2 方法里面执行 method1 时当前线程又要去尝试获取锁，这 时如果不支持重入，它就要等释放，把自己阻塞，导致自己锁死自己。

对 Synchronized 来说，可重入性是显而易见的，刚才提到，在执行 monitorenter 指令时，如果这个对象没有锁定，或者当前线程已经拥有了这个对象的锁(而不是已拥有了锁则不能继续获取)，就把锁的计 数器 +1，其实本质上就通过这种方式实现了可重入性。

4、JVM 对 Java 的原生锁做了哪些优化?

在 Java 6 之前，Monitor 的实现完全依赖底层操作系统的互斥锁来 实现，也就是我们刚才在问题二中所阐述的获取/释放锁的逻辑。

由于 Java 层面的线程与操作系统的原生线程有映射关系，如果要将一 个

线程进行阻塞或唤起都需要操作系统的协助，这就需要从用户态切换 到内核态来执行，这种切换代价十分昂贵，很耗处理器时间，现代 JDK 中做了大量的优化。 一种优化是使用自旋锁，即在把线程进行阻塞操作之前先让线程自旋等 待一段时间，可能在等待期间其他线程已经解锁，这时就无需再让线程 执行阻塞操作，避免了用户态到内核态的切换。
现代 JDK 中还提供了三种不同的 Monitor 实现，也就是三种不同的 锁:
• 偏向锁(Biased Locking)
• 轻量级锁
• 重量级锁
这三种锁使得 JDK 得以优化 Synchronized 的运行，当 JVM 检测 到不同的竞争状况时，会自动切换到适合的锁实现，这就是锁的升级、 降级。
• 当没有竞争出现时，默认会使用偏向锁。
JVM 会利用 CAS 操作，在对象头上的 Mark Word 部分设置线程 ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁，因 为在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定， 使用偏斜锁可以降低无竞争开销。
• 如果有另一线程试图锁定某个被偏斜过的对象，JVM 就撤销偏斜锁， 切换到轻量级锁实现。
• 轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁，如果重试成功， 就使用普通的轻量级锁;否则，进一步升级为重量级锁。

5、为什么说 Synchronized 是非公平锁?

非公平主要表现在获取锁的行为上，并非是按照申请锁的时间前后给等 待线程分配锁的，每当锁被释放后，任何一个线程都有机会竞争到锁， 这样做的目的是为了提高执行性能，缺点是可能会产生线程饥饿现象。

6、什么是锁消除和锁粗化?

• 锁消除:指虚拟机即时编译器在运行时，对一些代码上要求同步，但被 检测到不可能存在共享数据竞争的锁进行消除。主要根据逃逸分析。
程序员怎么会在明知道不存在数据竞争的情况下使用同步呢?很多不是 程序员自己加入的。
• 锁粗化:原则上，同步块的作用范围要尽量小。但是如果一系列的连续 操作都对同一个对象反复加锁和解锁，甚至加锁操作在循环体内，频繁 地进行互斥同步操作也会导致不必要的性能损耗。
锁粗化就是增大锁的作用域。

7、为什么说 Synchronized 是一个悲观锁?乐观锁的实现原理 又是什么?

什么是 CAS，它有什么特性?

Synchronized 显然是一个悲观锁，因为它的并发策略是悲观的: 不管是否会产生竞争，任何的数据操作都必须要加锁、用户态核心态转 换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略。先进行操作，如果没有其他线程征用数据，那操作就成功了; 如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种乐观的并发策略的许多实现不需要线程挂起，所以被称为非阻塞同步。乐观锁的核心算法是CAS(Compareand Swap，比较并交换)，它涉及到三个操作数:内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。

这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一 样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操 作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此 块内存。CAS 具有原子性，它的原子性由 CPU 硬件指令实现保证，即使用 JNI 调用 Native 方法调用由 C++ 编写的硬件级别指令，JDK 中提 供了 Unsafe 类执行这些操作。

8、乐观锁一定就是好的吗?

乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，但它也有缺点:

1. 乐观锁只能保证一个共享变量的原子操作。如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。

2. 长时间自旋可能导致开销大。假如 CAS 长时间不成功而一直自旋，会给 CPU 带来很大的开销。

3. ABA 问题。CAS 的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是 A， 后来被一条线程改为 B，最后又被改成了 A，则 CAS 认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入版本号，每次变量更新都把版本号加一。

9、跟 Synchronized 相比，可重入锁 ReentrantLock 其实现 原理有什么不同?

其实，锁的实现原理基本是为了达到一个目的: 让所有的线程都能看到某种标记。

Synchronized 通过在对象头中设置标记实现了这一目的，是一种 JVM 原生的锁实现方式，而 ReentrantLock 以及所有的基于 Lock 接口的 实现类，都是通过用一个 volitile 修饰的 int 型变量，并保证每个线 程都能拥有对该 int 的可见性和原子修改，其本质是基于所谓的 AQS 框架。

10、那么请谈谈 AQS 框架是怎么回事儿?

AQS(AbstractQueuedSynchronizer 类)是一个用来构建锁和同步器的框架，各种 Lock 包中的锁(常用的有 ReentrantLock、 ReadWriteLock) ，以及其他如 Semaphore、CountDownLatch， 甚至是早期的 FutureTask 等，都是基于 AQS 来构建。

1. AQS 在内部定义了一个 volatile int state 变量，表示同步状态:当线程调用 lock 方法时 ，如果 state=0，说明没有任何线程占有共享资源 的锁，可以获得锁并将 state=1;如果 state=1，则说明有线程目前正在 使用共享变量，其他线程必须加入同步队列进行等待。
2. AQS 通过 Node 内部类构成的一个双向链表结构的同步队列，来完成线程获取锁的排队工作，当有线程获取锁失败后，就被添加到队列末尾。
   • Node 类是对要访问同步代码的线程的封装，包含了线程本身及其状态叫
   waitStatus(有五种不同取值，分别表示是否被阻塞，是否等待唤醒， 是否已经被取消等)，每个 Node 结点关联其 prev 结点和 next 结 点，方便线程释放锁后快速唤醒下一个在等待的线程，是一个 FIFO 的过程。
   • Node 类有两个常量，SHARED 和 EXCLUSIVE，分别代表共享模式和独占模式。所谓共享模式是一个锁允许多条线程同时操作(信号量 Semaphore 就是基于 AQS 的共享模式实现的)，独占模式是同一个时间段只能有一个线程对共享资源进行操作，多余的请求线程需要排队等待 (如 ReentranLock) 。
3. AQS 通过内部类 ConditionObject 构建等待队列(可有多个)，当 Condition 调用 wait() 方法后，线程将会加入等待队列中，而当 Condition 调用 signal() 方法后，线程将从等待队列转移动同步队列中进行锁竞争。
4. AQS 和 Condition 各自维护了不同的队列，在使用 Lock 和 Condition 的时候，其实就是两个队列的互相移动。

11、请尽可能详尽地对比下 Synchronized 和 ReentrantLock 的异同。

ReentrantLock 是 Lock 的实现类，是一个互斥的同步锁。从功能角度，ReentrantLock 比 Synchronized 的同步操作更精细(因为可以像普通对象一样使用)，甚至实现 Synchronized 没有的高级功能，如:

• 等待可中断:当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，对处理执行时间非常长的同步块很有用。

• 带超时的获取锁尝试:在指定的时间范围内获取锁，如果时间到了仍然无法获取则返回。

• 可以判断是否有线程在排队等待获取锁。

• 可以响应中断请求:与 Synchronized 不同，当获取到锁的线程被中断时，能够响应中断，中断异常将会被抛出，同时锁会被释放。

• 可以实现公平锁。

从锁释放角度，Synchronized 在 JVM 层面上实现的，不但可以通过一些监控工具监控 Synchronized 的锁定，而且在代码执行出现异常 时，JVM 会自动释放锁定;但是使用 Lock 则不行，Lock 是通过代码实现的，要保证锁定一定会被释放，就必须将 unLock() 放到 finally{} 中 。

从性能角度，Synchronized 早期实现比较低效，对比ReentrantLock，大多数场景性能都相差较大。

但是在 Java 6 中对其进行了非常多的改进，在竞争不激烈时，Synchronized 的性能要优于 ReetrantLock;在高竞争情况下，Synchronized 的性能会下降几十倍，但是 ReetrantLock 的性能能维持常态。

12、ReentrantLock 是如何实现可重入性的?

ReentrantLock 内部自定义了同步器 Sync(Sync 既实现了 AQS， 又实现了 AOS，而 AOS 提供了一种互斥锁持有的方式)，其实就是 加锁的时候通过 CAS 算法，将线程对象放到一个双向链表中，每次获 取锁的时候，看下当前维护的那个线程 ID 和当前请求的线程 ID 是否 一样，一样就可重入了。

13、除了 ReetrantLock，你还接触过 JUC 中的哪些并发工具?

通常所说的并发包(JUC)也就是 java.util.concurrent 及其子包，集中了 Java 并发的各种基础工具类，具体主要包括几个方面:

• 提供了 CountDownLatch、CyclicBarrier、Semaphore等，比 Synchronized 更加高级，可以实现更加丰富多线程操作的同步结构。

• 提供了 ConcurrentHashMap、有序的 ConcunrrentSkipListMap，或者通过类似快照机制实现线程安全的动态数组 CopyOnWriteArrayList 等各种线

程安全的容器。
• 提供了 ArrayBlockingQueue、 SynchorousQueue 或针对特定场景的 PriorityBlockingQueue 等，各种并发队列实现。
• 强大的 Executor 框架，可以创建各种不同类型的线程池，调度任务运行等。

14、请谈谈 ReadWriteLock 和 StampedLock。

虽然 ReentrantLock 和 Synchronized 简单实用，但是行为上有一定局限性，要么不占，要么独占。实际应用场景中，有时候不需要大量 竞争的写操作，而是以并发读取为主，为了进一步优化并发操作的粒 度，Java 提供了读写锁。读写锁基于的原理是多个读操作不需要互斥，如果读锁试图锁定时，写锁是被某个线程持有，读锁将无法获得，而只好等待对方操作结束，这样就可以自动保证不会读取到有争议的数据。

ReadWriteLock 代表了一对锁，下面是一个基于读写锁实现的数据结 构，当数据量较大，并发读多、并发写少的时候，能够比纯同步版本凸 显出优势:

```
public class RWSample {
 private final Map<String, String> m = new TreeMap<>();
 private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
 private final Lock r = rwl.readLock();
 private final Lock w = rwl.writeLock();
 public String get(String key) {
     r.lock();
     System.out.println(" 读锁锁定! ");
     try {
         return m.get(key);
     } finally {
         r.unlock();
     }
 }

 public String put(String key, String entry) {
     w.lock();
     System.out.println(" 写锁锁定! ");
     try {
          return m.put(key, entry);
     } finally {
          w.unlock();
     }
 }
 // …
}
```

读写锁看起来比 Synchronized 的粒度似乎细一些，但在实际应用 中，其表现也并不尽如人意，主要还是因为相对比较大的开销。所以，JDK 在后期引入了 StampedLock，在提供类似读写锁的同时，还支持优化读模式。优化读基于假设，大多数情况下读操作并不会和写 操作冲突，其逻辑是先试着修改，然后通过 validate 方法确认是否进入了写模式，如果没有进

入，就成功避免了开销;如果进入，则尝试获取读锁。

```
public class StampedSample {
 private final StampedLock sl = new StampedLock();

 void mutate() {
     long stamp = sl.writeLock();
     try {
         write();
     } finally {
         sl.unlockWrite(stamp);
     }
 }

 Data access() {
     long stamp = sl.tryOptimisticRead();
     Data data = read();
     if (!sl.validate(stamp)) {
         stamp = sl.readLock();
         try {
             data = read();
         } finally {
             sl.unlockRead(stamp);
         }
     }
     return data;
 }
 // -
}
```

15、如何让 Java 的线程彼此同步?你了解过哪些同步器?请分别 介绍下。

JUC 中的同步器三个主要的成员:CountDownLatch、CyclicBarrier 和 Semaphore，通过它们可以方便地实现很多线程之间协作的功能。

CountDownLatch 叫倒计数，允许一个或多个线程等待某些操作完成。看几个场景:

• 跑步比赛，裁判需要等到所有的运动员(“其他线程”)都跑到终点 (达到目标)，才能去算排名和颁奖。

• 模拟并发，我需要启动 100 个线程去同时访问某一个地址，我希望它 们能同时并发，而不是一个一个的去执行。

用法:CountDownLatch 构造方法指明计数数量，被等待线程调用 countDown 将计数器减 1，等待线程使用 await 进行线程等待。一 个简单的例子:

```java
public class TestCountDownLatch {
    private CountDownLatch countDownLatch = new CountDownLatch(4); // 构造方法指明计数数量

    public static void main(String[] args) {
        TestCountDownLatch testCountDownLatch = new TestCountDownLatch();
        testCountDownLatch.begin();
    }
    // 运动员类
    private class Runner implements Runnable {
        private int result;
        public Runner(int result) {
            this.result = result;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(result * 1000); // 模拟跑了多少秒
                countDownLatch.countDown();  // 跑完了就计数器减1
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void begin() {
        System.out.println("赛跑开始");
        Random random = new Random(System.currentTimeMillis());
        for (int i = 0; i < 4; i++) {
            int result = random.nextInt(3) + 1; // 随机设置每个运动员跑多少秒结束
            new Thread(new Runner(result)).start();
        }
        try {
            countDownLatch.await(); // 线程等待倒数为0
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("所有人都跑完了，裁判开始算成绩");
    }
```

CyclicBarrier 叫循环栅栏，它实现让一组线程等待至某个状态之后再全部同时执行，而且当所有等待线程被释放后，CyclicBarrier 可以被 重复使用。CyclicBarrier 的典型应用场景是用来等待并发线程结束。CyclicBarrier 的主要方法是 await()，await() 每被调用一次，计数便 会减少 1，并阻塞住当前线程。当计数减至 0 时，阻塞解除，所有在 此 CyclicBarrier 上面阻塞的线程开始运行。

在这之后，如果再次调用 await()，计数就又会变成 N-1，新一轮重新开始，这便是 Cyclic 的含义所在。CyclicBarrier.await() 带有返回值，用来表示当前线程是第几个到达这个 Barrier 的线程。

举例说明如下:

```java
public class TestCyclicBarrier {
    private CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) {
        new TestCyclicBarrier().begin();
    }

    public void begin() {
        for (int i = 0; i < 5; i++) {
            new Thread(new Student()).start();
        }

    }

    private class Student implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(2000);     // 该学生正在赶往饭店的路上
                cyclicBarrier.await(); // 到了就等着，等其他人都到了，就进饭店
            } catch (Exception e) {
                e.printStackTrace();
            }
            // TODO:大家都到了，进去吃饭吧!
        }
    }
}
```

Semaphore，Java 版本的信号量实现，用于控制同时访问的线程个数，来达到限制通用资源访问的目的，其原理是通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

```java
public class Test {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(5); // 机器数目，即5个许可
        for(int i = 0; i < 8; i++)              // 工人数，8个去抢许可
            new Worker(i,semaphore).start();
    }

    static class Worker extends Thread{
        private int num;
        private Semaphore semaphore;
        public Worker(int num,Semaphore semaphore){
            this.num = num;
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                semaphore.acquire(); // 抢许可
                Thread.sleep(2000);
                semaphore.release(); // 释放许可
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

如果 Semaphore 的数值被初始化为1，那么一个线程就可以通过 acquire 进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比

如互斥锁是有持有者的，而对于 Semaphore 这种计数器结构，虽然有类似功能，但其实不存在真正意义的持有者，除非我们进行扩展包装。

16、CyclicBarrier 和 CountDownLatch 看起来很相似，请对比 下呢?

它们的行为有一定相似度，区别主要在于:

• CountDownLatch 是不可以重置的，所以无法重用，CyclicBarrier 没有这种限制，可以重用。

• CountDownLatch 的基本操作组合是 countDown/await，调用 await 的线程阻塞等待 countDown 足够的次数，不管你是在一个线程还是多个线程里 countDown，只要次数足够即可。 CyclicBarrier 的基本操作组合就是 await，当所有的伙伴都调用了 await，才会继续进行任务，并自动进行重置。

CountDownLatch 目的是让一个线程等待其他 N 个线程达到某个条件后，自己再去做某个事(通过 CyclicBarrier 的第二个构造方法 public CyclicBarrier(int parties, Runnable barrierAction)，在新线程里做事可以达到同样的效果)。而 CyclicBarrier 的目的是让 N 多 线程互相等待直到所有的都达到某个状态，然后这 N 个线程再继续执行各自后续(通过 CountDownLatch 在某些场合也能完成类似的效果)。

17、Java 中的线程池是如何实现的?

• 在 Java 中，所谓的线程池中的“线程”，其实是被抽象为了一个静态 内部类 Worker，它基于 AQS 实现，存放在线程池的 HashSet workers 成员变量中;

• 而需要执行的任务则存放在成员变量 workQueue(BlockingQueue workQueue)中。

这样，整个线程池实现的基本思想就是:从 workQueue 中不断取出 需要执行的任务，放在 Workers 中进行处理。

18、创建线程池的几个核心构造参数?

Java 中的线程池的创建其实非常灵活，我们可以通过配置不同的参 数，创建出行为不同的线程池，这几个参数包括:

• corePoolSize:线程池的核心线程数。

• maximumPoolSize:线程池允许的最大线程数。

• keepAliveTime:超过核心线程数时闲置线程的存活时间。

• workQueue:任务执行前保存任务的队列，保存由 execute 方法提交的 Runnable 任务 。

19、线程池中的线程是怎么创建的?是一开始就随着线程池的启动 创建好的吗?

显然不是的。线程池默认初始化后不启动 Worker，等待有请求时才启动。

每当我们调用 execute() 方法添加一个任务时，线程池会做如下判 断:

- 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务;
- 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列;
- 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务;
- 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。

当一个线程完成任务时，它会从队列中取下一个任务来执行。 当一个线程无事可做，超过一定的时间(keepAliveTime)时，线程池会判断。

如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

20、既然提到可以通过配置不同参数创建出不同的线程池，那么 Java 中默认实现好的线程池又有哪些呢?请比较它们的异同。

1. SingleThreadExecutor 线程池

这个线程池只有一个核心线程在工作，也就是相当于单线程串行执行所 有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来 替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- corePoolSize:1，只有一个核心线程在工作。
- maximumPoolSize: 1。
- keepAliveTime: 0L。
- `workQueue:new LinkedBlockingQueue<Runnable>()`，其缓冲队列是无界的。

2. FixedThreadPool 线程池

FixedThreadPool 是固定大小的线程池，只有核心线程。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

FixedThreadPool 多数针对一些很稳定很固定的正规并发线程，多用于服

务器。
• corePoolSize: nThreads
• maximumPoolSize: nThreads
• keepAliveTime: 0L
• `workQueue:new LinkedBlockingQueue<Runnable>()`，其缓冲队列是无界的。
3. CachedThreadPool 线程池
CachedThreadPool 是无界线程池，如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲(60秒不执行任务)线程，当 任务数增加时，此线程池又可以智能的添加新线程来处理任务。线程池大小完全依赖于操作系统(或者说 JVM)能够创建的最大线程大小。SynchronousQueue 是一个是缓冲区为 1 的阻塞队列。缓存型池子通常用于执行一些生存期很短的异步型任务，因此在一些面向连接的 daemon 型 SERVER 中用得不多。但对于生存期短的异步任务，它是 Executor 的首选。
• `corePoolSize: 0`
• `maximumPoolSize: Integer.MAX_VALUE`
• `keepAliveTime: 60L`
• `workQueue:new SynchronousQueue<Runnable>()`，一个是缓冲区为 1 的阻塞队列。
4. `ScheduledThreadPool` 线程池
`ScheduledThreadPool`:核心线程池固定，大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。创建一个周期性执行任务的线程池。如果闲置，非核心线程池会在 `DEFAULT_KEEPALIVEMILLIS` 时间内回收。
• `corePoolSize: corePoolSize`
• `maximumPoolSize: Integer.MAX_VALUE`
• `keepAliveTime: DEFAULT_KEEPALIVE_MILLIS`
• `workQueue:new DelayedWorkQueue()`

21、如何在 Java 线程池中提交线程?

线程池最常用的提交任务的方法有两种:
1. execute(): ExecutorService.execute 方法接收一个例，它用来执行一个任务:

```
ExecutorService.execute(Runnable runable)
```

2. submit(): ExecutorService.submit() 方法返回的是 Future 对象。可以用

isDone() 来查询 Future 是否已经完成，当任务完成时， 它具有一个结果，可以调用 get() 来获取结果。也可以不用 isDone() 进行检查就直接调用 get()，在这种情况下，get() 将阻塞，直至结果准备就绪。

```
FutureTask task = ExecutorService.submit(Runnable runnable);
FutureTask<T> task = ExecutorService.submit(Runnable runnable,T Result);
FutureTask<T> task = ExecutorService.submit(Callable<T> callable);
```

22、什么是 Java 的内存模型，Java 中各个线程是怎么彼此看到 对方的变量的?

Java 的内存模型定义了程序中各个变量的访问规则，即在虚拟机中将 变量存储到内存和从内存中取出这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法参数，因为这些是线程私有的，不会被共享，所以不存在竞争问题。

Java 中各个线程是怎么彼此看到对方的变量的呢?Java 中定义了主内 存与工作内存的概念:

所有的变量都存储在主内存，每条线程还有自己的工作内存，保存了被 该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作(读取、赋值)都必须在工作内存中进行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

23、请谈谈 volatile 有什么特点，为什么它能保证变量对所有线 程的可见性?

关键字 volatile 是 Java 虚拟机提供的最轻量级的同步机制。当一个 变量被定义成 volatile 之后，具备两种特性:

1. 保证此变量对所有线程的可见性。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。

2. 禁止指令重排序优化。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

Java 的内存模型定义了 8 种内存间操作:

lock 和 unlock

- 把一个变量标识为一条线程独占的状态。
- 把一个处于锁定状态的变量释放出来，释放之后的变量才能被其他线程锁定。

read 和 write

- 把一个变量值从主内存传输到线程的工作内存，以便 load。
- 把 store 操作从工作内存得到的变量的值，放入主内存的变量中。

load 和 store

• 把 read 操作从主内存得到的变量值放入工作内存的变量副本中。 • 把工作内存的变量值传送到主内存，以便 write。

use 和 assgin

• 把工作内存变量值传递给执行引擎。

• 将执行引擎值传递给工作内存变量值。

volatile 的实现基于这 8 种内存间操作，保证了一个线程对某个 volatile 变量的修改，一定会被另一个线程看见，即保证了可见性。

24、既然 volatile 能够保证线程间的变量可见性，是不是就意味 着基于 volatile 变量的运算就是并发安全的?

显然不是的。基于 volatile 变量的运算在并发下不一定是安全的。 volatile 变量在各个线程的工作内存，不存在一致性问题(各个线程的工作内存中 volatile 变量，每次使用前都要刷新到主内存)。

但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的。

25、请对比下 volatile 对比 Synchronized 的异同。

Synchronized 既能保证可见性，又能保证原子性，而 volatile 只能保证可见性，无法保证原子性。

ThreadLocal 和 Synchonized 都用于解决多线程并发访问，防止任务在共享资源上产生冲突。但是 ThreadLocal 与 Synchronized 有本质的区别。

Synchronized 用于实现同步机制，是利用锁的机制使变量或代码块在某一时该只能被一个线程访问，是一种 “以时间换空间” 的方式。

而 ThreadLocal 为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，根除了对变量的共享，是一种 “以空间换时间” 的方式。

26、请谈谈 ThreadLocal 是怎么解决并发安全的?

ThreadLocal 这是 Java 提供的一种保存线程私有信息的机制，因为 其在整个线程生命周期内有效，所以可以方便地在一个线程关联的不同业务模块之间传递信息，比如事务 ID、Cookie 等上下文相关信息。

ThreadLocal 为每一个线程维护变量的副本，把共享数据的可见范围限 制在同一个线程之内，其实现原理是，在 ThreadLocal 类中有一个 Map，用于存储每一个线程的变量的副本。

27、很多人都说要慎用 ThreadLocal，谈谈你的理解，使用

ThreadLocal 需要注意些什么?

使用 ThreadLocal 要注意 remove!

ThreadLocal 的实现是基于一个所谓的 ThreadLocalMap，在 ThreadLocalMap 中，它的 key 是一个弱引用。

通常弱引用都会和引用队列配合清理机制使用，但是 ThreadLocal 是个例外，它并没有这么做。

这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应 ThreadLocalMap!这就是很多 OOM 的来源，所 以通常都会建议，应用一定要自己负责 remove，并且不要和线程池配 合，因为 worker 线程往往是不会退出的。

四、开源框架面试题专栏

4.1、Spring面试整理

1、什么是 Spring 框架?Spring 框架有哪些主要模块?

Spring 框架是一个为 Java 应用程序的开发提供了综合、广泛的基础性支持的 Java 平台。 Spring 帮助开发者解决了开发中基础性的问题，使得开发人员可以专注于应用程序的开发。

Spring 框架本身亦是按照设计模式精心打造，这使得我们可以在开发环境中安心的集成 Spring 框架，不必担心 Spring 是如何在后台进行工作的。

Spring 框架至今已集成了 20 多个模块。这些模块主要被分如下图所示的核心容器、数据访问/集成,、Web、AOP(面向切面编程)、工具、消息和测试模块。

2、使用 Spring 框架能带来哪些好处?

下面列举了一些使用 Spring 框架带来的主要好处:

• Dependency Injection(DI) 方法使得构造器和 JavaBean properties 文件中的依赖关系一目了然。

• 与 EJB 容器相比较，IoC 容器更加趋向于轻量级。这样一来 IoC 容器在有限的内存和 CPU 资源的情况下进行应用程序的开发和发布就变得十分有利。

• Spring 并没有闭门造车，Spring 利用了已有的技术比如 ORM 框架、logging 框架、J2EE、Q uartz和JDK Timer，以及其他视图技术。

• Spring 框架是按照模块的形式来组织的。由包和类的编号就可以看出其所属的模块，开发者仅仅需要选用他们需要的模块即可。

• 要测试一项用 Spring 开发的应用程序十分简单，因为测试相关的环境代码都已经囊括在框架中了。更加简单的是，利用 JavaBean 形式的 POJO 类，可以很方便的利用依赖注入来写入测试数据。
• Spring 的 Web 框架亦是一个精心设计的 Web MVC 框架，为开发者们在 web 框架的选择上提供了一个除了主流框架比如 Struts、过度设计的、不流行 web 框架的以外的有力选项。
• Spring 提供了一个便捷的事务管理接口，适用于小型的本地事物处理(比如在单 DB 的环境下)和复杂的共同事物处理(比如利用 JTA 的复杂 DB 环境)。

3、什么是控制反转(IOC)?什么是依赖注入?

控制反转是应用于软件工程领域中的，在运行时被装配器对象来绑定耦合对象的一种编程技巧，对象之间耦合关系在编译时通常是未知的。在传统的编程方式中，业务逻辑的流程是由应用程序中的早已被设定好关联关系的对象来决定的。在使用控制反转的情况下，业务逻辑的流程是由对象关系图来决定的，该对象关系图由装配器负责实例化，这种实现方式还可以将对象之间的关联关系的定义抽象化。而绑定的过程是通过“依赖注入”实现的。
控制反转是一种以给予应用程序中目标组件更多控制为目的设计范式，并在我们的实际工作中起到了有效的作用。
依赖注入是在编译阶段尚未知所需的功能是来自哪个的类的情况下，将其他对象所依赖的功能对象实例化的模式。这就需要一种机制用来激活相应的组件以提供特定的功能，所以依赖注入是控制反转的基础。否则如果在组件不受框架控制的情况下，框架又怎么知道要创建哪个组件?
在 Java 中依然注入有以下三种实现方式:
1. 构造器注入
2. Setter 方法注入
3. 接口注入

4、请解释下 Spring 框架中的 IoC?

Spring 中的 org.springframework.beans 包和 org.springframework.context 包
构成了 Spring 框架 IoC 容器的基础。
BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。 ApplicationContex 接口对 BeanFactory(是一个子接口)进行了扩展，在 BeanFactory 的基础上添加了其他功能，比如与 Spring 的

AOP 更容易集成，也提供了处理 message resource 的机制(用于国际化)、事件传播以及应用层的特别配置，比如针对 Web 应用的 WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是 Spring IoC 容器的具体实现，用来包装和管理前面提到的各种 bean。BeanFactory 接口是 Spring IoC 容器的核心接口。

IOC:把对象的创建、初始化、销毁交给 spring 来管理，而不是由开发者控制，实现控制反转。

5、BeanFactory 和 ApplicationContext 有什么区别?

BeanFactory 可以理解为含有 bean 集合的工厂类。BeanFactory 包含了种 bean 的定义，以便在接收到客户端请求时将对应的 bean 实例化。
BeanFactory 还能在实例化对象的时生成协作类之间的关系。此举将 bean 自身与 bean 客户端的 配置中解放出来。BeanFactory 还包含了 bean 生命周期的控制，调用客户端的初始化方法 (initialization methods)和销毁方法(destruction methods)。
从表面上看，application context 如同 bean factory 一样具有 bean 定义、bean 关联关系的设置，根据请求分发 bean 的功能。但 applicationcontext 在此基础上还提供了其他的功能。
1. 提供了支持国际化的文本消息
2. 统一的资源文件读取方式
3. 已在监听器中注册的bean的事件
以下是三种较常见的 ApplicationContext 实现方式:
1、ClassPathXmlApplicationContext:从 classpath 的 XML 配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中

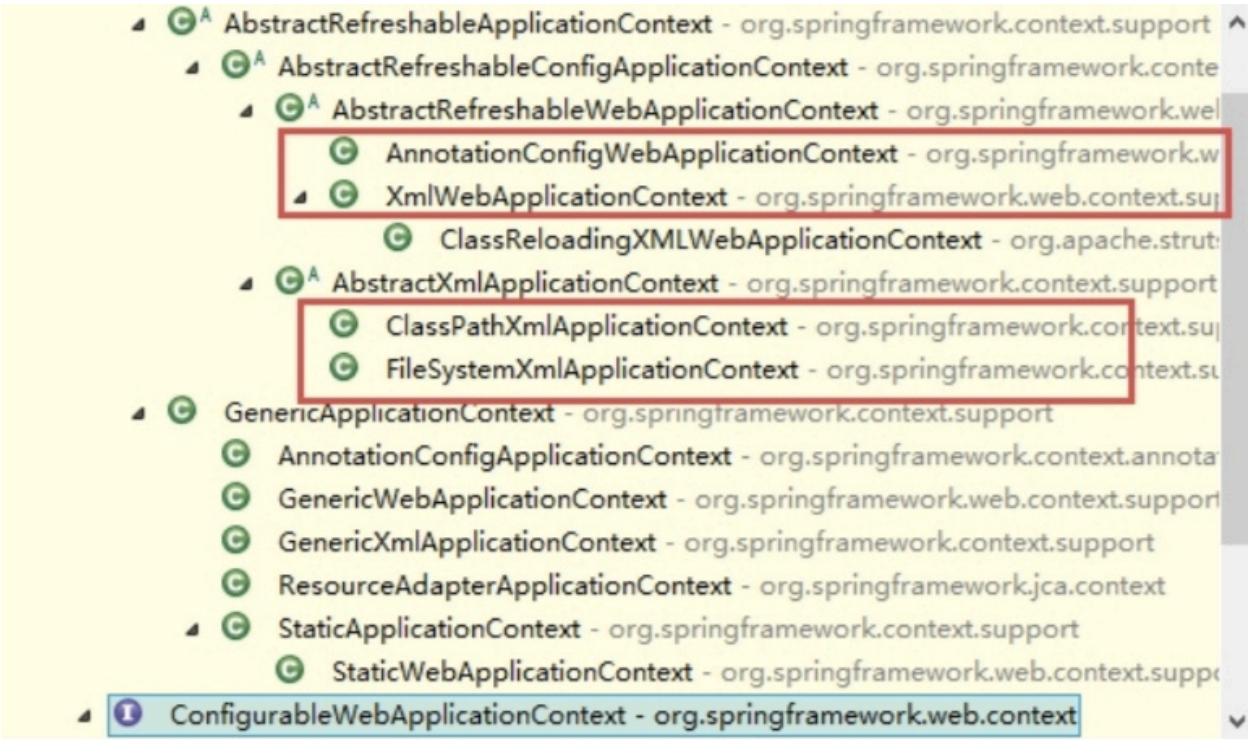
```
ApplicationContext context = new ClassPathXmlApplicationContex
t(“bean.xml”);
```

2、FileSystemXmlApplicationContext :由文件系统中的 XML 配置文件读取上下文。

```
ApplicationContext context = new FileSystemXmlApplicationConte
xt(“bean.xml”);
```

3、XmlWebApplicationContext:由 Web 应用的 XML 文件读取上下文。

4、AnnotationConfigApplicationContext(基于 Java 配置启动容器)



6、Spring 有几种配置方式?

将 Spring 配置到应用开发中有以下三种方式:

1. 基于XML的配置
2. 基于注解的配置
3. 基于Java的配置

7、如何用基于 XML 配置的方式配置 Spring?

在 Spring 框架中，依赖和服务需要在专门的配置文件来实现，我常用的 XML 格式的配置文件。这 些配置文件的格式通常用 `<beans>` 开头，然后一系列的 bean 定义和专门的应用配置选项组成。

SpringXML 配置的主要目的时候是使所有的 Spring 组件都可以用 xml 文件的形式来进行配置。这意味着不会出现其他的 Spring 配置类型(比如声明的方式或基于 Java Class 的配置方式)

Spring 的 XML 配置方式是使用被 Spring 命名空间的所支持的一系列的 XML 标签来实现的。Spring 有以下主要的命名空间:context、beans、jdbc、tx、aop、mvc 和 aso。

如:

```
<beans>
    <!-- JSON Support -->
    <bean name="viewResolver"
class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
    <bean name="jsonTemplate"
class="org.springframework.web.servlet.view.json.MappingJackson2JsonV
iew"/>
    <bean id="restTemplate"
class="org.springframework.web.client.RestTemplate"/>
</beans>
```

下面这个 web.xml 仅仅配置了 DispatcherServlet，这件最简单的配置便能满足应用程序配置运行时组件的需求。

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

8、如何用基于 Java 配置的方式配置 Spring?

pring 对 Java 配置的支持是由@Configuration 注解和@Bean 注解来实现的。由@Bean 注解的方法将会实例化、配置和初始化一个 新对象，这个对象将由 Spring 的 IoC 容器来管理。@Bean 声明所起到的作用与 `<bean/>` 元素类似。被 @Configuration 所注解的类则表示这个类的主要目的是作为 bean 定义的资源。被@Configuration 声明的类可以通过在同一个类的内部调 用@bean 方法来设置嵌入 bean 的依赖关系。

最简单的@Configuration 声明类请参考下面的代码:

```
@Configuration
public class AppConfig{
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

对于上面的@Beans 配置文件相同的 XML 配置文件如下:

```
<beans>
    <bean id="myService" class="com.somnus.services.MyServiceI
mpl"/>
</beans>
```

上述配置方式的实例化方式如下:利用 AnnotationConfigApplicationContext 类进行实例化

```
public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
    }
```

要使用组件组建扫描，仅需用@Configuration 进行注解即可:

```
@Configuration
@ComponentScan(basePackages = "com.somnus")
public class AppConfig  {
... }
```

在上面的例子中，com.acme 包首先会被扫到，然后再容器内查找被 @Component 声明的类，找到后将这些类按照 Sring bean 定义进行注册。

如果你要在你的 web 应用开发中选用上述的配置的方式的话，需要用 AnnotationConfigWebApplicationContext 类来读取配置文件，可以用来配

置 Spring 的 Servlet 监听器 ContextLoaderListener 或者 Spring MVC 的 DispatcherServlet。

```
<web-app>
    <!-- Configure ContextLoaderListener to use
AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
org.springframework.web.context.support.AnnotationConfigWebApp
licatio
nContext
        </param-value>
    </context-param>
<!-- Configuration locations must consist of one or more comma
- or space-delimited
        fully-qualified @Configuration classes. Fully-qualifie
d
packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.howtodoinjava.AppConfig</param-value>
    </context-param>
    <!-- Bootstrap the root application context as usual using
ContextLoaderListener -->
<web-app>
    <!-- Configure ContextLoaderListener to use
AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
org.springframework.web.context.support.AnnotationConfigWebApp
```

```
licatio
nContext
        </param-value>
    </context-param>
<!-- Configuration locations must consist of one or more comma
- or space-delimited
        fully-qualified @Configuration classes. Fully-qualifie
d
packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.howtodoinjava.AppConfig</param-value>
    </context-param>
    <!-- Bootstrap the root application context as usual using
ContextLoaderListener -->
```

9、怎样用注解的方式配置 Spring?

Spring 在 2.5 版本以后开始支持用注解的方式来配置依赖注入。可以用注解的方式来替代 XML 方式的 bean 描述，可以将 bean 描述转移到组件类的内部，只需要在相关类上、方法上或者字段声明上使用注解即可。注解注入将会被容器在 XML 注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结 果。

注解装配在 Spring 中是默认关闭的。所以需要在 Spring 文件中配置一下才能使用基于注解的装配模式。如果你想要在你的应用程序中使用关于注解的方法的话，请参考如下的配置。

```
<beans>
   <context:annotation-config/>
   <!-- bean definitions go here -->
</beans>
```

在 `<context:annotation-config/>` 标签配置完成以后，就可以用注解的方式在 Spring 中向属性、方法和构造方法中自动装配变量。

下面是几种比较重要的注解类型:

1. @Required:该注解应用于设值方法。

2. @Autowired:该注解应用于有值设值方法、非设值方法、构造方法和变量。
3. @Qualifier:该注解和@Autowired 注解搭配使用，用于消除特定 bean 自动装配的歧义。
4. JSR-250 Annotations:Spring 支持基于 JSR-250 注解的以下注解，@Resource、 @PostConstruct 和 @PreDestroy。

10、请解释 Spring Bean 的生命周期?

Spring Bean 的生命周期简单易懂。在一个 bean 实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个 bean 不在被调用时需要进行相关的析构操作，并从 bean 容 器中移除。
Spring bean factory 负责管理在 spring 容器中被创建的 bean 的生命周期。Bean 的生命周期由两组回调(call back)方法组成。
1. 初始化之后调用的回调方法。
2. 销毁之前调用的回调方法。
Spring 框架提供了以下四种方式来管理 bean 的生命周期事件:
• InitializingBean 和 DisposableBean 回调接口
• 针对特殊行为的其他 Aware 接口
• Bean配置文件中的Custom init()方法和destroy()方法
• @PostConstruct 和@PreDestroy 注解方式
使用 customInit()和 customDestroy()方法管理 bean 生命周期的代码样例如下:

```
<beans>
    <bean id="demoBean" class="com.somnus.task.DemoBean" init-method="customInit" destroy-method="customDestroy"></bean>
</beans>
```

11、Spring Bean 的作用域之间有什么区别?

Spring 容器中的 bean 可以分为 5 个范围。所有范围的名称都是自说明的，但是为了避免混淆，还是让我们来解释一下:
1. singleton:这种 bean 范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个 bean 的实例，单例的模式由 bean factory 自身来维护。
2. prototype:原形范围与单例范围相反，为每一个 bean 请求提供一个实例。

3. request:在请求 bean 范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean 会失效并被垃圾回收器回收。
4. Session:与请求范围类似，确保每个 session 中有一个 bean 的实例，在 session 过期后，bean 会随之失效。
5. global- session:global-session 和 Portlet 应用相关。当你的应用部署在 Portlet 容器中工作时，它包含很多 portlet。如果你想要声明让所有的 portlet 共用全局的存储变量的话，那么这全局变量需要存储在 global-session 中。
全局作用域与 Servlet 中的 session 作用域效果相同。

12、什么是 Spring inner beans?

在 Spring 框架中，无论何时 bean 被使用时，当仅被调用了一个属性。一个明智的做法是将这个 bean 声明为内部 bean。内部 bean 可以用 setter 注入“属性”和构造方法注入“构造参数”的方式来 实现。
比如，在我们的应用程序中，一个 Customer 类引用了一个 Person 类，我们的要做的是创建一个 Person 的实例，然后在 Customer 内部使用。

```
public class Customer{
    private Person person;
    //Setters and Getters
    }
    public class Person{
    private String name;
    private String address;
    private int age;
    //Setters and Getters
}
```

内部 bean 的声明方式如下:

```
<bean id="CustomerBean" class="com.somnus.common.Customer">
    <property name="person">
        <!-- This is inner bean -->
        <bean class="com.howtodoinjava.common.Person">
            <property name="name" value="lokesh" />
            <property name="address" value="India" />
            <property name="age" value="34" />
```

```
        </bean>
    </property>
</bean>
```

13、Spring 框架中的单例 Beans 是线程安全的么?

Spring 框架并没有对单例 bean 进行任何多线程的封装处理。关于单例 bean 的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的 Spring bean 并没有可变的状态(比如 Serview 类和 DAO 类)，所以在某种程度上说 Spring 的单例 bean 是线程安全的。如果你的 bean 有多种状态的话(比如 View Model 对象)，就需要自行保证线程安全。

最浅显的解决办法就是将多态 bean 的作用域由“singleton”变更为“prototype”。

14、请举例说明如何在 Spring 中注入一个 Java Collection?

Spring 提供了以下四种集合类的配置元素:

- `<list>` : 该标签用来装配可重复的 list 值。
- `<set>` : 该标签用来装配没有重复的 set 值。
- `<map>` : 该标签可用来注入键和值可以为任何类型的键值对。
- `<props>` : 该标签支持注入键和值都是字符串类型的键值对。

下面看一下具体的例子:

```
<beans>
<!-- Definition for javaCollection -->
<bean id="javaCollection" class="com.howtodoinjava.JavaCollect
ion">
      <!-- java.util.List -->
      <property name="customList">
        <list>
           <value>INDIA</value>
           <value>Pakistan</value>
           <value>USA</value>
           <value>UK</value>
        </list>
      </property>
     <!-- java.util.Set -->
     <property name="customSet">
```

```
        <set>
           <value>INDIA</value>
           <value>Pakistan</value>
           <value>USA</value>
           <value>UK</value>
        </set>
      </property>
     <!-- java.util.Map -->
     <property name="customMap">
        <map>
           <entry key="1" value="INDIA"/>
           <entry key="2" value="Pakistan"/>
           <entry key="3" value="USA"/>
           <entry key="4" value="UK"/>
</map>
</property>
    <!-- java.util.Properties -->
    <property name="customProperies">
        <props>
            <prop key="admin">admin@nospam.com</prop>
            <prop key="support">support@nospam.com</prop>
        </props>
</property>
   </bean>
</beans>
```

15、如何向 Spring Bean 中注入一个 Java.util.Properties?

第一种方法是使用如下面代码所示的 `<props>` 标签:

```
<bean id="adminUser" class="com.somnus.common.Customer">
    <!-- java.util.Properties -->
    <property name="emails">
        <props>
            <prop key="admin">admin@nospam.com</prop>
            <prop key="support">support@nospam.com</prop>
        </props>
```

```
    </property>
</bean>
```

也可用”util:”命名空间来从 properties 文件中创建出一个 propertiesbean，然后利用 setter 方 法注入 bean 的引用。

16、请解释 Spring Bean 的自动装配?

在 Spring 框架中，在配置文件中设定 bean 的依赖关系是一个很好的机制，Spring 容器还可以自 动装配合作关系 bean 之间的关联关系。这意味着 Spring 可以通过向 Bean Factory 中注入的方 式自动搞定 bean 之间的依赖关系。自动装配可以设置在每个 bean 上，也可以设定在特定的 bean 上。

下面的 XML 配置文件表明了如何根据名称将一个 bean 设置为自动装配:

```
<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImp
l"
autowire="byName" />
```

除了 bean 配置文件中提供的自动装配模式，还可以使用@Autowired 注解来自动装配指定 的 bean。在使用@Autowired 注解之前需要在按照如下的配置方式在 Spring 配置文件进行配置才可以使用。

```
<context:annotation-config />
```

也可以通过在配置文件中配置 AutowiredAnnotationBeanPostProcessor 达到相同的效果。

```
<bean class
="org.springframework.beans.factory.annotation.AutowiredAnnota
tionBea
nPostProcessor"/>
```

配置好以后就可以使用@Autowired 来标注了。

```
@Autowired
public EmployeeDAOImpl ( EmployeeManager manager ) {
    this.manager = manager;
}
```

17、请解释自动装配模式的区别?

在 Spring 框架中共有 5 种自动装配，让我们逐一分析。

1. no:这是 Spring 框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在 bean 定义 中用标签明确的设置依赖关系。

2. byName:该选项可以根据 bean 名称设置依赖关系。当向一个 bean 中自动装配一个属性时，容器将根据 bean 的名称自动在在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

3. byType:该选项可以根据 bean 类型设置依赖关系。当向一个 bean 中自动装配一个属性时，容器将根据 bean 的类型自动在在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

4. constructor:造器的自动装配和 byType 模式类似，但是仅仅适用于与有构造器相同参数的 bean，如果在容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常。

5. autodetect:该模式自动探测使用构造器自动装配或者 byType 自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在 bean 内部没有找到相应的构造器或者是无参构造器，容器就会自动选择 byTpe 的自动装配方式。

18、如何开启基于注解的自动装配?

要使用 @Autowired，需要注册 AutowiredAnnotationBeanPostProcessor，可以

有以下两种方式来实现:

1、引入配置文件中的 `<bean>` 下引入 `<context:annotation-config>`

```
<beans>
    <context:annotation-config />
</beans>
```

2、在 bean 配置文件中直接引入 AutowiredAnnotationBeanPostProcessor

```
<beans>
    <bean
class="org.springframework.beans.factory.annotation.AutowiredA
nnotati
onBeanPostProcessor"/>
```

```
</beans>
```

19、请举例解释@Required 注解?

在产品级别的应用中，IoC 容器可能声明了数十万了 bean，bean 与 bean 之间有着复杂的依赖关系。设值注解方法的短板之一就是验证所有的属性是否被注解是一项十分困难的操作。可以通过在 `<bean>` 中设置“dependency-check”来解决这个问题。

在应用程序的生命周期中，你可能不大愿意花时间在验证所有 bean 的属性是否按照上下文件正确配置。或者你宁可验证某个 bean 的特定属性是否被正确的设置。即使是用“dependency- check”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解。

需要用如下的方式使用来标明 bean 的设值方法。

```
public class EmployeeFactoryBean extends AbstractFactoryBean<O
bject>{
    private String designation;
    public String getDesignation() {
        return designation;
    }
    @Required
    public void setDesignation(String designation) {
        this.designation = designation;
    }
    //more code here
}
```

RequiredAnnotationBeanPostProcessor 是 Spring 中的后置处理用来验证被 @Required 注解的 bean 属性是否被正确的设置了。在使用 RequiredAnnotationBeanPostProcesso 来验证 bean 属性之前，首先要在 IoC 容器中对其进行注册:

```
<bean
class="org.springframework.beans.factory.annotation.RequiredAn
notatio
nBeanPostProcessor" />
```

但是如果没有属性被用 @Required 注解过的话，后置处理器会抛出一个

BeanInitializationException 异常。

20、请举例解释@Autowired 注解?

@Autowired 注解对自动装配何时何处被实现提供了更多细粒度的控制。@Autowired 注解可 以像@Required 注解、构造器一样被用于在 bean 的设值方法上自动装配 bean 的属性，一个参数或者带有任意名称或带有多个参数的方法。比如，可以在设值方法上使用@Autowired 注解来替代配置文件中的 `<property>` 元素。当 Spring 容器在 setter 方法上找到 @Autowired 注解时，会尝试用 byType 自动装配。

当然我们也可以在构造方法上使用@Autowired 注解。带有@Autowired 注解的构造方法意味着 在创建一个 bean 时将会被自动装配，即便在配置文件中使用 元素。

```
public class TextEditor {
   private SpellChecker spellChecker;
   @Autowired
   public TextEditor(SpellChecker spellChecker){
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck(){
      spellChecker.checkSpelling();
} }
```

下面是没有构造参数的配置方式:

```
<beans>
   <context:annotation-config/>
   <!-- Definition for textEditor bean without constructor-arg
  -->
   <bean id="textEditor" class="com.howtodoinjava.TextEditor"/
>
   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.howtodoinjava.SpellCheck
er"/>
   </beans>
```

21、请举例说明@Qualifier 注解?

@Qualifier 注解意味着可以在被标注 bean 的字段上可以自动装配。

Qualifier 注解可以用来取消 Spring 不能取消的 bean 应用。

下面的示例将会在 Customer 的 person 属性中自动装配 person 的值。

```
public class Customer{
    @Autowired
    private Person person;
}
```

下面我们要在配置文件中来配置 Person 类。

```
<bean id="customer" class="com.somnus.common.Customer" />
<bean id="personA" class="com.somnus.common.Person" >
    <property name="name" value="lokesh" />
</bean>
<bean id="personB" class="com.somnus.common.Person" >
    <property name="name" value="alex" />
</bean>
```

Spring 会知道要自动装配哪个 person bean 么?不会的，但是运行上面的示例时，会抛出下面的异常:

```
Caused by:
org.springframework.beans.factory.NoSuchBeanDefinitionExceptio
n:
No unique bean of type [com.howtodoinjava.common.Person] is de
fined: expected single matching bean but found 2: [personA, pe
rsonB]
```

要解决上面的问题，需要使用 @Quanlifier 注解来告诉 Spring 容器要装配哪个 bean:

```
public class Customer{
    @Autowired
    @Qualifier("personA")
    private Person person;
}
```

22、构造方法注入和设值注入有什么区别? 请注意以下明显的区别:

1. 在设值注入方法支持大部分的依赖注入，如果我们仅需要注入int、string和long型的变量，我们不要用设值的方法注入。对于基本类型，如果我们没有注入的话，可以为基本类型设置默认值。在构造方法 注入不支持大部分的依赖注入，因为在调用构造方法中必须传入正确的构造参数，否则的话为报错。

2. 设值注入不会重写构造方法的值。如果我们对同一个变量同时使用了构造方法注入又使用了设置方法注入的话，那么构造方法将不能覆盖由设值方法注入的值。很明显，因为构造方法尽在对象被创建时调用。

3. 在使用设值注入时有可能还不能保证某种依赖是否已经被注入，也就是说这时对象的依赖关系有可能是不完整的。而在另一种情况下，构造器注入则不允许生成依赖关系不完整的对象。

4. 在设值注入时如果对象A和对象B互相依赖，在创建对象A时Spring会抛出 sObjectCurrentlyInCreationException 异常，因为在 B 对象被创建之前 A 对 象是不能被创建的，反之亦然。所以 Spring 用设值注入的方法解决了循环依赖的问题，因对象的设值方法是在对象被创建之前被调用的。

23、Spring 框架中有哪些不同类型的事件?

Spring 的 ApplicationContext 提供了支持事件和代码中监听器的功能。我们可以创建 bean 用来监听在 ApplicationContext 中发布的事件。ApplicationEvent 类和在 ApplicationContext 接口中处理的事件，如果一个 bean 实现了 ApplicationListener 接口，当一个 ApplicationEvent 被发布以后，bean 会自动被通知。

```
public class AllApplicationEventListener implements Applicatio
nListener < ApplicationEvent >{
    @Override
    public void onApplicationEvent(ApplicationEvent applicatio
nEvent)
    {
        //process event
    }
}
```

Spring 提供了以下 5 中标准的事件:

1. 上下文更新事件(ContextRefreshedEvent):该事件会在ApplicationContext

被初始化或者更新时发布。也可以在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。
2. 上下文开始事件(ContextStartedEvent):当容器调用 ConfigurableApplicationContext的 Start()方法开始/重新开始容器时触发该事件。
3. 上下文停止事件(ContextStoppedEvent):当容器调用 ConfigurableApplicationContext的 Stop()方法停止容器时触发该事件。
4. 上下文关闭事件(ContextClosedEvent):当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
5. 请求处理事件(RequestHandledEvent):在Web应用中，当一个http请求(request)结束 触发该事件。
除了上面介绍的事件以外，还可以通过扩展 ApplicationEvent 类来开发自定义的事件。

```
public class CustomApplicationEvent extends ApplicationEvent{
public CustomApplicationEvent ( Object source, final String ms
g ){
        super(source);
        System.out.println("Created a Custom event");
    }
}
```

为了监听这个事件，还需要创建一个监听器:

```
public class CustomEventListener implements ApplicationListene
r <
CustomApplicationEvent >{
@Override
    public void onApplicationEvent(CustomApplicationEvent
applicationEvent) {
        //handle event
    }
}
```

之后通过 applicationContext 接口的 publishEvent()方法来发布自定义事件。

```
CustomApplicationEvent customEvent = new
CustomApplicationEvent(applicationContext, "Test message");
applicationContext.publishEvent(customEvent);
```

24、FileSystemResource 和 ClassPathResource 有何区别?

在 FileSystemResource 中需要给出 spring-config.xml 文件在你项目中的相对路径或者绝对路径。在 ClassPathResource 中 spring 会在 ClassPath 中自动搜寻配置文件，所以要把 ClassPathResource 文件放在 ClassPath 下。

如果将 spring-config.xml 保存在了 src 文件夹下的话，只需给出配置文件的名称即可，因为 src 文件夹是默认。

简而言之，ClassPathResource 在环境变量中读取配置文件，FileSystemResource 在配置文件中读取配置文件。

25、Spring 框架中都用到了哪些设计模式?

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的:

o 代理模式—在AOP和remoting中被用的比较多。

o 单例模式—在spring配置文件中定义的bean默认为单例模式。

o 模板方法—用来解决代码重复的问题。比如.RestTemplate,JmsTemplate,JpaTempl ate。

o 前端控制器—Spring提供了DispatcherServlet来对请求进行分发。

o 视图帮助(ViewHelper)—Spring提供了一系列的JSP标签，高效宏来辅助将分散的代码整合在视图里。

o 依赖注入—贯穿于BeanFactory/ApplicationContext接口的核心理念。

o 工厂模式—BeanFactory用来创建对象的实例

26、开发中主要使用 Spring 的什么技术 ?

1. IOC 容器管理各层的组件
2. 使用 AOP 配置声明式事务
3. 整合其他框架.

27、简述 AOP 和 IOC 概念 AOP:

Aspect Oriented Program, 面向(方面)切面的编程;Filter(过滤器) 也是一种 AOP. AOP 是一种新的方法论, 是对传统 OOP(Object-Oriented Programming, 面向对象编程) 的补充. AOP 的 主要编程对象是切面(aspect), 而切面模块化横切关注点.可以举例通过事务说明.

IOC: Invert Of Control, 控制反转. 也成为 DI(依赖注入)其思想是反转资源获

取的方向. 传统的资源查找方式要求组件向容器发起请求查找资源.作为回应, 容器适时的返回资源. 而应用了 IOC 之后, 则是容器主动地将资源推送给它所管理的组件,组件所要做的仅是选择一种合适的方式来接受资源. 这种行为也被称为查找的被动形式

28、在 Spring 中如何配置 Bean ?

Bean 的配置方式: 通过全类名(反射)、通过工厂方法(静态工厂方法 & 实例工厂方法)、FactoryBean

29、IOC 容器对 Bean 的生命周期:

1. 通过构造器或工厂方法创建 Bean 实例
2. 为 Bean 的属性设置值和对其他 Bean 的引用
3. 将 Bean 实例传递给 Bean 后置处理器的 postProcessBeforeInitialization 方法
4. 调用 Bean 的初始化方法(init-method)
5. 将 Bean 实例传递给 Bean 后置处理器的 postProcessAfterInitialization 方法
6. Bean 可以使用了
7. 当容器关闭时, 调用 Bean 的销毁方法(destroy-method)

4.2 SpringMVC面试整理

1、什么是 SpringMvc?

SpringMvc 是 spring 的一个模块，基于 MVC 的一个框架，无需中间整合层来整合。

2、Spring MVC 的优点:

1.它是基于组件技术的.全部的应用对象,无论控制器和视图,还是业务对象之类的都是 java 组件.并且和 Spring 提供的其他基础结构紧密集成.
2.不依赖于 Servlet API(目标虽是如此,但是在实现的时候确实是依赖于 Servlet 的)
3.可以任意使用各种视图技术,而不仅仅局限于 JSP
4.支持各种请求资源的映射策略
5.它应是易于扩展的

3、SpringMVC 工作原理?

1.客户端发送请求到 DispatcherServlet
2.DispatcherServlet 查询 handlerMapping 找到处理请求的 Controller

3.Controller 调用业务逻辑后，返回 ModelAndView
4.DispatcherServlet 查询 ModelAndView，找到指定视图
5.视图将结果返回到客户端

4、SpringMVC 流程?

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

6、SpringMvc 的控制器是不是单例模式,如果是,有什么问题,怎么解决?

是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

7、如果你也用过 struts2.简单介绍下 springMVC 和 struts2 的区别有哪些?

1.springmvc 的入口是一个 servlet 即前端控制器，而 struts2 入口是一个 filter 过虑器。
2.springmvc 是基于方法开发(一个 url 对应一个方法)，请求参数传递到方法的形参，可以 设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设 计为多例。
3.Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 reques 域传输到页面。Jsp 视图解析器默认使用 jstl。

8、SpingMvc 中的控制器的注解一般用那个,有没有别的注解可以替代?
一般用@Conntroller 注解,表示是表现层,不能用用别的注解代替。

9、 @RequestMapping 注解用在类上面有什么作用?
是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

10、怎么样把某个请求映射到特定的方法上面?
答:直接在方法上面加上注解@RequestMapping,并且在这个注解里面写上要拦截的路径

11、如果在拦截请求中,我想拦截 get 方式提交的方法,怎么配置?
可以在@RequestMapping 注解里面加上 method=RequestMethod.GET

12、怎么样在方法里面得到 Request,或者 Session?
直接在方法的形参中声明 request,SpringMvc 就自动把 request 对象传入

13、我想在拦截的方法里面得到从前台传入的参数,怎么得到?
答:直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样

14、如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?
直接在方法中声明这个对象,SpringMvc 就自动会把属性赋值到这个对象里面。

15、SpringMvc 中函数的返回值是什么?
答:返回值可以有很多类型,有 String, ModelAndView,当一般用 String 比较好。

16、SpringMVC 怎么样设定重定向和转发的?
在返回值前面加"forward:"就可以让结果转发,譬如"forward:user.do?name=method4" 在返回值前面加"redirect:"就可以让返回值重定向,譬如 `"redirect:http://www.baidu.com"`

17、SpringMvc 用什么对象从后台向前台传递数据的?
答:通过 ModelMap 对象,可以在这个对象里面用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。

18、SpringMvc 中有个类把视图和数据都合并的一起的,叫什么?
叫 ModelAndView。

19、怎么样把 ModelMap 里面的数据放入 Session 里面?

可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key

20、SpringMvc 怎么和 AJAX 相互调用的?

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。具体步骤如下 :

1.加入 Jackson.jar

2.在配置文件中配置 json 的映射

3.在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上 @ResponseBody 注解

21、当一个方法向 AJAX 返回特殊对象,譬如 Object,List 等,需要做什么处理?

要加上@ResponseBody 注解

22、SpringMvc 里面拦截器是怎么写的

有两种写法,一种是实现接口,另外一种是继承适配器类,然后在 SpringMvc 的配置文件中 配置拦截器即可:

```
<!-- 配置 SpringMvc 的拦截器 --> <mvc:interceptors>
<!-- 配置一个拦截器的 Bean 就可以了 默认是对所有请求都拦截 -->
<bean id="myInterceptor" class="com.et.action.MyHandlerInterce
ptor"></bean> <!-- 只针对部分请求拦截 -->
<mvc:interceptor>
<mvc:mapping path="/modelMap.do" />
<bean class="com.et.action.MyHandlerInterceptorAdapter" /> </m
vc:interceptor>
</mvc:interceptors>
```

23、讲下 SpringMvc 的执行流程

系统启动的时候根据配置文件创建spring的容器, 首先是发送http请求到核心控制器 disPatherServlet，spring 容器通过映射器去寻找业务控制器，使用适配器找到相应的业务类，在进业务类时进行数据封装，在封装前可能会涉及到类型转换，执行完业务类后使用 ModelAndView 进行视图转发，数据放在 model 中，用 map 传递数据进行页面显示。

4.3、MyBatis面试整理

1、什么是 MyBatis?

MyBatis 是一个可以自定义 SQL、存储过程和高级映射的持久层框架。

2、讲下 MyBatis 的缓存

MyBatis 的缓存分为一级缓存和二级缓存,一级缓存放在 session 里面,默认就有,二级缓 存放在它的命名空间里,默认是不打开的,使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置 `<cache/>`

3、Mybatis 是如何进行分页的?分页插件的原理是什么?

1.Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 现分页，也可以使用 Mybatis 的分页插件。

2.分页插件的原理:实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。

举例:

```
select * from student
```

拦截 sql 后重写为:

```
select t.* from (select * from student)t limit 0, 10
```

4、简述 Mybatis 的插件运行原理，以及如何编写一个插件?

1.Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、 Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实 现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

2.实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

5、Mybatis 动态 sql 是做什么的?都有哪些动态 sql?能简述一下动态 sql 的执行原理吗?

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

2.Mybatis 提供了 9 种动态 sql 标签:

trim|where|set|foreach|if|choose|when|otherwise|bind。
3.其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

6、#{}和${}的区别是什么?

1. `#{}` 是预编译处理， `${}` 是字符串替换。
2.Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法
来赋值;
3.Mybatis 在处理 `${}` 时，就是把 `${}` 替换成变量的值。
4.使用 `#{}` 可以有效的防止 SQL 注入，提高系统安全性。

7、为什么说 Mybatis 是半自动 ORM 映射工具?它与全自动的区别在哪里?

Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

8、Mybatis 是否支持延迟加载?如果支持，它的实现原理是什么?

1.Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否 启用延迟加载
`lazyLoadingEnabled=true|false` 。
2.它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 `a.getB().getName()` ，拦截器 invoke() 方法发现 a.getB()是 null 值，那么就会单 独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 `a.setB(b)` ，于是 a 的 对象 b 属性就有值了，接着完成 `a.getB().getName()` 方法的调用。这就是延迟加载的基本原理。

9、MyBatis 与 Hibernate 有哪些不同?

1.Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 java 对象。
2.Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控

制 sql 执行性 能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定 义多套 sql 映射文件，工作量大。

3.Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件(例如需求固定的定制化软件)如果用 hibernate 开发可以节省很多代码，提高效率。但是 Hibernate 的缺点是学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。 总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

10、MyBatis 的好处是什么?

1.MyBatis 把 sql 语句从 Java 源程序中独立出来，放在单独的 XML 文件中编写，给程序的维护带来了很大便利。

2.MyBatis 封装了底层 JDBC API 的调用细节，并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作。

3.因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活 控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。

11、简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系?

Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中， `<parameterMap>` 标签会被解析为 ParameterMap 对象，其每个子元素会 被解析为 ParameterMapping 对象。 `<resultMap>` 标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个 `<select>`、`<insert>`、`<update>`、`<delete>` 标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。

12、什么是 MyBatis 的接口绑定,有什么好处?

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

13、接口绑定有几种实现方式,分别是怎么实现的?

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上 @Select@Update 等注解里面包含 Sql 语句来绑定,另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名.

14、什么情况下用注解绑定,什么情况下用 xml 绑定?

当 Sql 语句比较简单时候,用注解绑定;当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多

15、MyBatis 实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成;嵌套查询是先查一个表,根据这个表里面的结果的外键 id,去再另外一个表里面查询数据,也是通过 association 配置,但另外一个表的查询通过 select 属性配置。

16、Mybatis 能执行一对一、一对多的关联查询吗?都有哪些实现方式,以及它们之间的区别?

能，Mybatis 不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把 selectOne()修改为 selectList()即可;多对多查询，其实就是一对多查询，只需要把 selectOne()修改为 selectList()即可。

关联对象查询，有两种实现方式，一种是单独发送一个 sql 去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用 join 查询，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。

17、MyBatis 里面的动态 Sql 是怎么设定的?用什么语法?

MyBatis 里面的动态 Sql 一般是通过 if 节点来实现,通过 OGNL 语法来实现,但是如果要写的完整,必须配合 where,trim 节点,where 节点是判断包含节点有内容就插入 where,否则不插入,trim 节点是用来判断如果动态语句是以 and 或 or 开始,那么会自动把这个 and 或者 or 取掉。

18、Mybatis 是如何将 sql 执行结果封装为目标对象并返回的?都有哪些映射形式?

第一种是使用 `<resultMap>` 标签，逐一定义列名和对象属性名之间的映

射关系。
第二种是使用 sql 列的别名功能，将列别名书写为对象属性名，比如 `T_NAME AS NAME`，对 象属性名一般是 name，小写，但是列名不区分大小写，Mybatis 会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成 `T_NAME AS NaMe, Mybatis` 一样可以正常工作。
有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

19、Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签?

还有很多其他的标签，`<resultMap>、<parameterMap>、<sql>、<include>、 <selectKey>`，加上动态 sql 的 9 个标签，`trim|where|set|foreach|if|choose|when|otherwise|bind` 等，其中 `<sql>` 为 sql 片段标签，通过 `<include>` 标签引入 sql 片段，`<selectKey>` 为不支持自增的主键生成策略标签。

20、当实体类中的属性名和表中的字段名不一样，如果将查询的结果封装到指定 pojo?

1.通过在查询的 sql 语句中定义字段名的别名。
2.通过 `<resultMap>` 来映射字段名和实体类属性名的一一对应的关系。

21、模糊查询 like 语句该怎么写

1.在 java 中拼接通配符，通过 `#{}` 赋值
2.在 Sql 语句中拼接通配符 (不安全会引起 Sql 注入)

22、通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应, Dao 的工作原理，是否可以重载?

不能重载，因为通过 Dao 寻找 Xml 对应的 sql 的时候全限名+方法名的保存和寻找策略。接口工作原理为 jdk 动态代理原理，运行时会为 dao 生成 proxy，代理对象会拦截接口方法，去执行对应的 sql 返回数据。

23、Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能 否定义在 A 标签的后面，还是说必须定义在 A 标签的前面?

虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不

存在，此时，Mybatis 会将 A 标签标记为未解 析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也就可以正常解析完成了。

24、Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复?

不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复;如果没有配置 namespace，那么 id 不能重复;毕竟 namespace 不是必须的，只是最佳实践而已。原因就是 namespace+id 是作为 `Map<String, MappedStatement>` 的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

25、Mybatis 中如何执行批处理?

使用 BatchExecutor 完成批处理。

26、Mybatis 都有哪些 Executor 执行器?它们之间的区别是什么?

Mybatis 有三种基本的 `Executor` 执行器， `SimpleExecutor、ReuseExecutor、 BatchExecutor。`

1.SimpleExecutor:每执行一次 `update` 或 `select`，就开启一个 `Statement` 对象，用完立刻关闭 Statement 对象。

2.ReuseExecutor:执行 update 或 select，以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后，不关闭 Statement 对象，而是放置于 Map

3.BatchExecutor:完成批处理。

27、Mybatis 中如何指定使用哪一种 Executor 执行器?

在 Mybatis 配置文件中，可以指定默认的 ExecutorType 执行器类型，也可以手动给 DefaultSqlSessionFactory 的创建 SqlSession 的方法传递 ExecutorType 类型参数。

28、Mybatis 执行批量插入，能返回数据库主键列表吗?

能，JDBC 都能，Mybatis 当然也能。

29、Mybatis 是否可以映射 Enum 枚举类?

Mybatis 可以映射枚举类，不单可以映射枚举类，Mybatis 可以映射任何对象到表的一列上。映射方式为自定义一个 TypeHandler，实现 TypeHandler

的 setParameter()和 getResult()接口方法。
TypeHandler 有两个作用，一是完成从 javaType 至 jdbcType 的转换， 二是完成 jdbcType 至 javaType 的转换，体现为 setParameter()和 getResult()两个方法，分别代表设置 sql 问号占位符参数和获取列查询结果。

30、如何获取自动生成的(主)键值?
配置文件设置 usegeneratedkeys 为 true

31、在 mapper 中如何传递多个参数?
答:
1.直接在方法中传递参数，xml 文件用#{0} #{1}来获取
2.使用 @param 注解:这样可以直接在 xml 文件中通过#{name}来获取

32、resultType resultMap 的区别?
1.类的名字和数据库相同时，可以直接设置 resultType 参数为 Pojo 类
2.若不同，需要设置 resultMap 将结果名字和 Pojo 名字进行转换

33、使用 MyBatis 的 mapper 接口调用时有哪些要求?
类型相同
1.Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同
2.Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql的 parameterType 的类型相同
3.Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同
4.Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

34、Mybatis 比 IBatis 比较大的几个改进是什么?
1.有接口绑定,包括注解绑定 sql 和 xml 绑定 Sql
2.动态 sql 由原来的节点配置变成 OGNL 表达式
3.在一对一,一对多的时候引进了association,在一对多的时候引入了 collection 节点,不过都是在 resultMap 里面配置

35、IBatis 和 MyBatis 在核心处理类分别叫什么?
IBatis 里面的核心处理类交 SqlMapClient,MyBatis 里面的核心处理类叫做 SqlSession。

36、IBatis 和 MyBatis 在细节上的不同有哪些?
1.在 sql 里面变量命名有原来的#变量# 变成了#{变量}
2.原来的 `$变量$` 变成了${变量}

3.原来在 sql 节点里面的 class 都换名字交 type
4.原来的queryForObject queryForList 变成了selectOne selectList5)原来的别名设置在映射文件里面放在了核心配置文件里

五、分布式面试专栏

5.1、分布式限流面试整理

5.1.1 ZooKeeper专题

1.ZooKeeper 是什么?

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。
客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid(Zookeeper Transaction Id)。而读请求只会相对于更新有序，也就是读请求的返回 结果中会带有这个 zookeeper 最新的 zxid。

2.ZooKeeper 提供了什么?

1、文件系统
2、通知机制

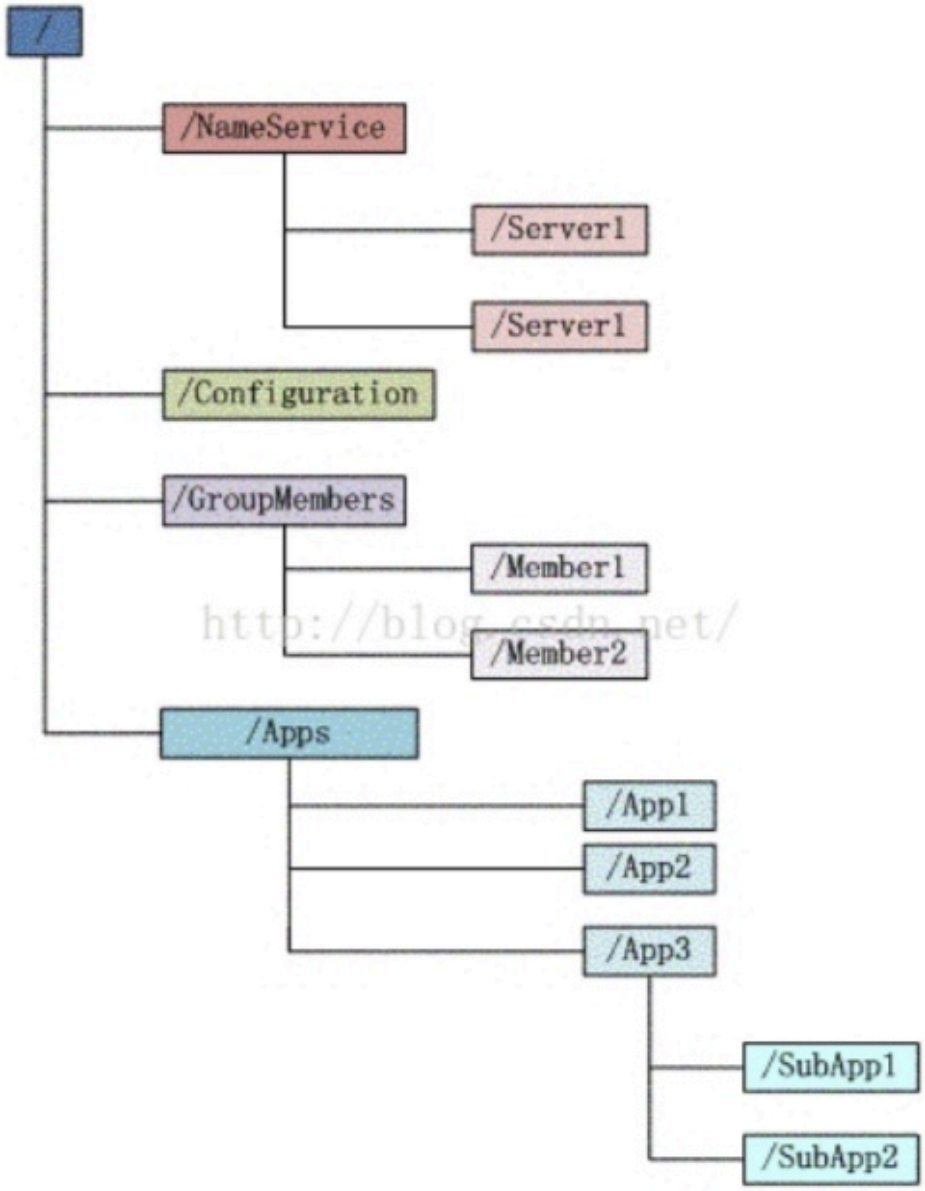
3.Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间(节点称为 znode)。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。

4.四种类型的 znode

1、 `PERSISTENT-` 持久化目录节点
客户端与 zookeeper 断开连接后，该节点依旧存在

2、 `PERSISTENT_SEQUENTIAL-` 持久化顺序编号目录节点
客户端与 zookeeper 断开连接后，该节点依旧存在，只是 Zookeeper 给该节点名称进行顺序编号
3、 `EPHEMERAL-` 临时目录节点
客户端与 zookeeper 断开连接后，该节点被删除
4、 `EPHEMERAL_SEQUENTIAL-` 临时顺序编号目录节点
客户端与 zookeeper 断开连接后，该节点被删除，只是 Zookeeper 给该节点名称进行顺序编号



5.Zookeeper 通知机制
client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

6.Zookeeper 做了什么?

1、命名服务
2、配置管理
3、集群管理
4、分布式锁
5、队列管理

7.zk 的命名服务(文件系统)

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

8.zk 的配置管理(文件系统、通知机制)

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

9.Zookeeper 集群管理(文件系统、通知机制)

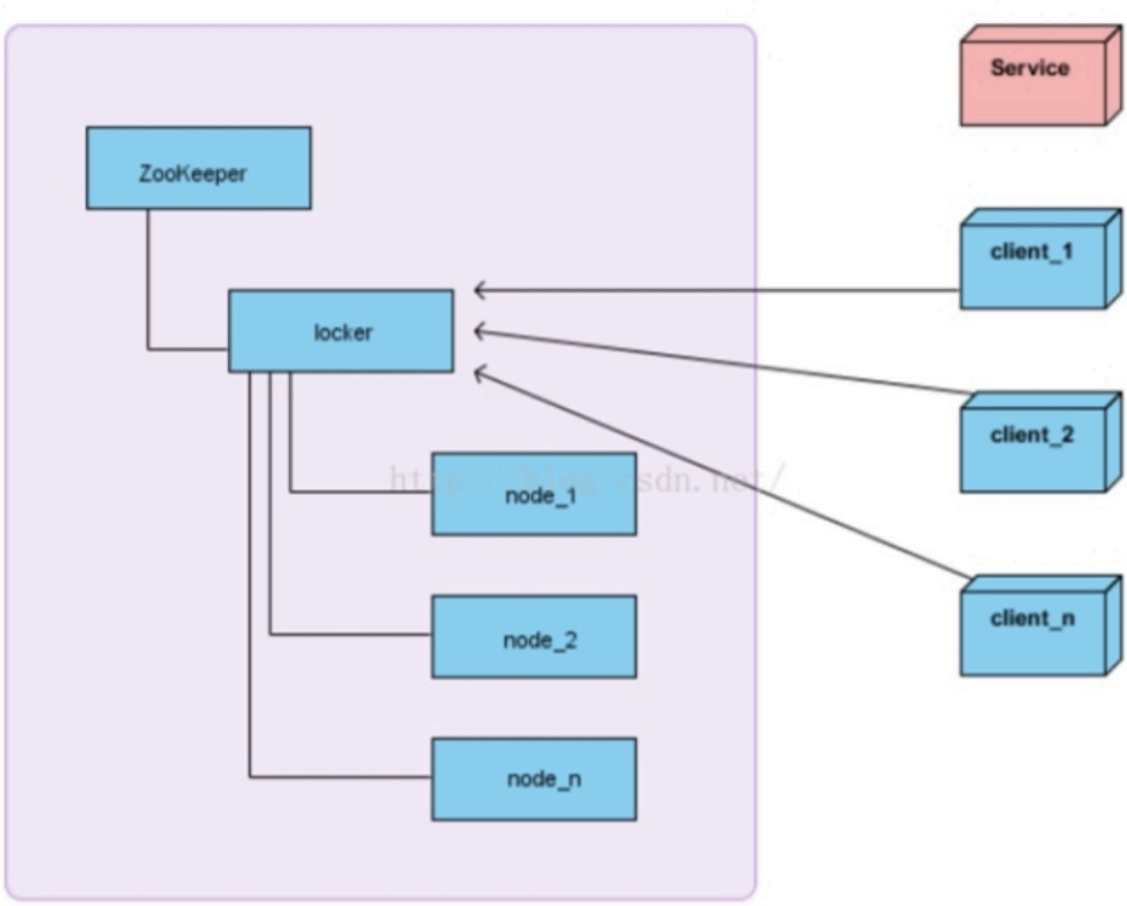
所谓集群管理无在乎两点:是否有机器退出和加入、选举 master。
对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知:某个兄弟目录被删除，于是，所有人都知道:它上船了。
新机器加入也是类似，所有机器收到通知:新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变 一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

10.Zookeeper 分布式锁(文件系统、通知 机制)

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控 制时序。
对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建 `/distribute_lock` 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 `distribute_lock` 节点就释放出锁。
对于第二类， `/distribute_lock` 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用

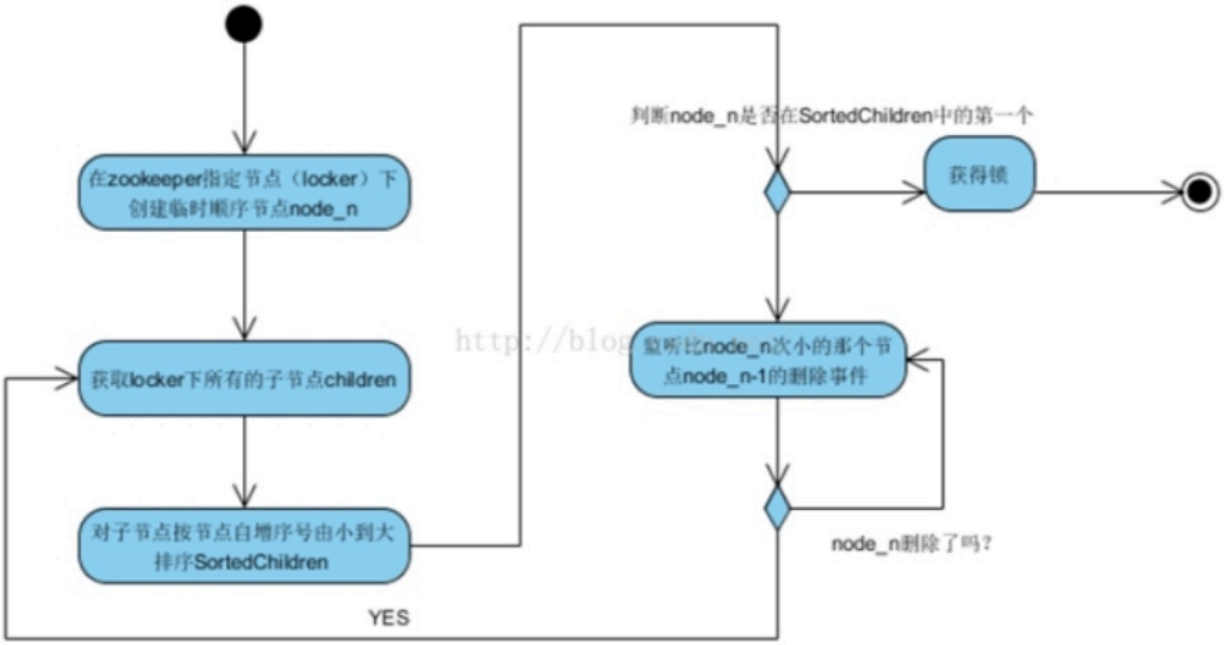
完删除，依次方便。

11.获取分布式锁的流程



在获取分布式锁的时候在 locker 节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用 createNode 方法在 locker 下创建临时顺序节点，

然后调用 getChildren(“locker”)来获取 locker 下面的所有子节点，注意此时不用设置任何 Watcher。客户端获取到所有的子节点 path 之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非 locker 所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用 exist()方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 locker 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。



代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于 BaseDistributedLock，实现了基于 Zookeeper 实现分布式锁的细节。

12.Zookeeper 队列管理(文件系统、通知机制)

两种类型的队列:

1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

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

13.Zookeeper 数据复制

Zookeeper 作为一个集群提供一致的数据服务，自然，它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，不致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，提高用户访问速度。
从客户端读写访问的透明度来看，数据复制集群系统分下面两种:
1、写主(WriteMaster) :对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离;
2、写任意(Write Any):对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。
对 zookeeper 来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降(这也是它建立 observer 的原因)，而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

14.Zookeeper 工作原理

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，它们分别是恢复模式(选主)和广播模式(同步)。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，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 leader 和 Server 具有相同的系统状态。

15.zookeeper 是如何保证事务的顺序一致性的?

zookeeper 采用了递增的事务 Id 来标识，所有的 proposal(提议)都在被提出的时候加上了 zxid，zxid 实际 上是一个 64 位的数字，高 32 位是 epoch(时期; 纪元; 世; 新时代)用来标识 leader 是否发生改变，如果有 新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

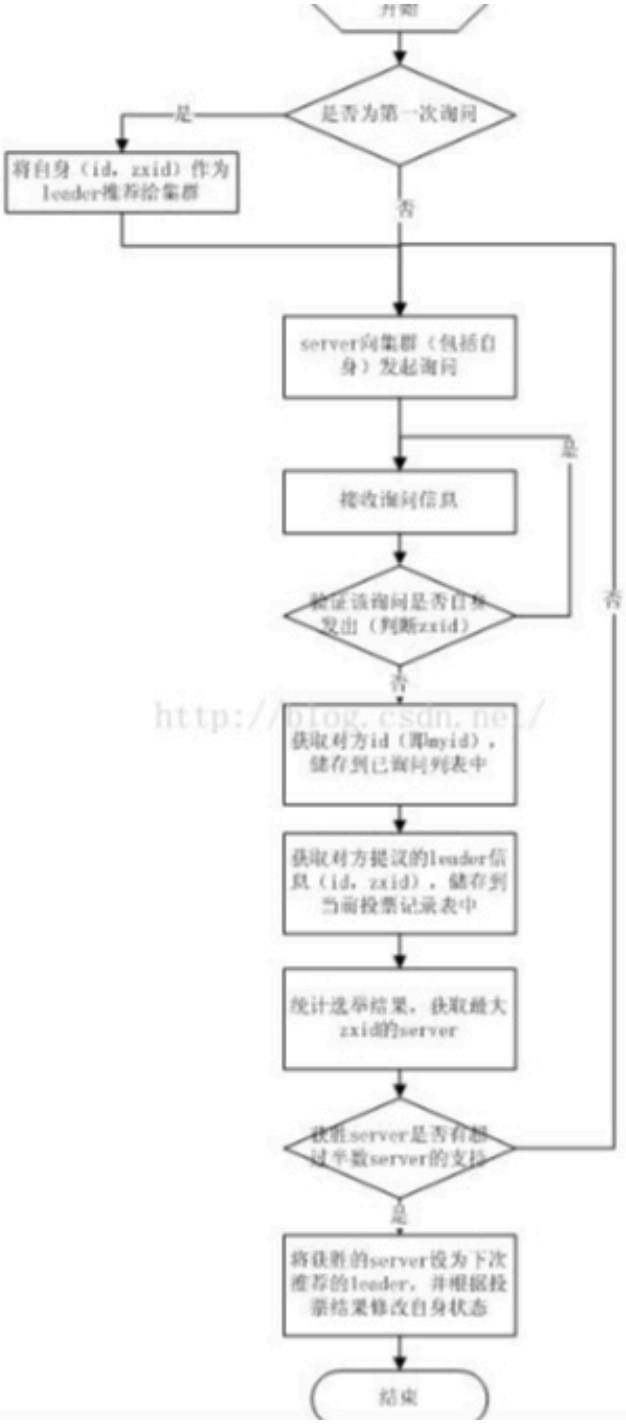
16.Zookeeper 下 Server 工作状态

每个 Server 在工作过程中有三种状态:
LOOKING:当前 Server 不知道 leader 是谁，正在搜寻 LEADING:当前 Server 即为选举出来的 leader FOLLOWING:leader 已经选举出来，当前 Server 与之同步

17.zookeeper 是如何选取主 leader 的?

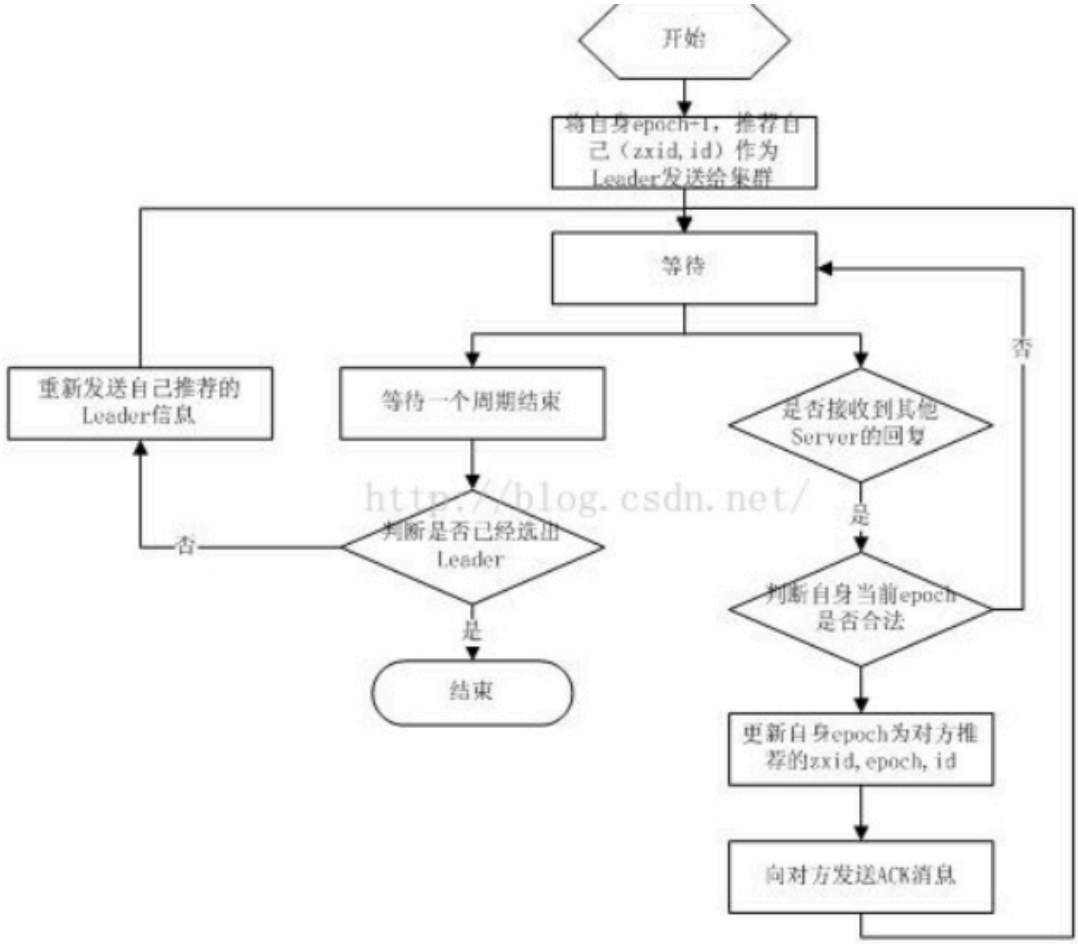
当 leader 崩溃或者 leader 失去大多数的 follower，这时 zk 进入恢复模

式，恢复模式需要重新选举出一个新的 leader，让所有的 Server 都恢复到一个正确的状态。Zk 的选举算法有两种:
一种是基于 basic paxos 实现的，
另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为 fast paxos。
1、Zookeeper 选主流程(basic paxos)
(1)选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server;
(2)选举线程首先向所有 Server 发起一次询问(包括自己);
(3)选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中;
(4)收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server;
(5)线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader，如果此时获胜的 Server 获得 n/2 + 1 的 Server 票数，设置当前推荐的 leader 为获胜的 Server，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 leader 被选举出来。 通过流程分析我们可以得出:要使 Leader 获得多数 Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1. 每个 Server 启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 server 还会从磁盘快照中恢复数据和会话信息，zk 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。



2、Zookeeper 选主流程(basic paxos)

fast paxos 流程是在选举过程中，某 Server 首先向所有 Server 提议自己要成为 leader，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出 Leader。



18.Zookeeper 同步流程

选完 Leader 以后，zk 就进入状态同步过程。

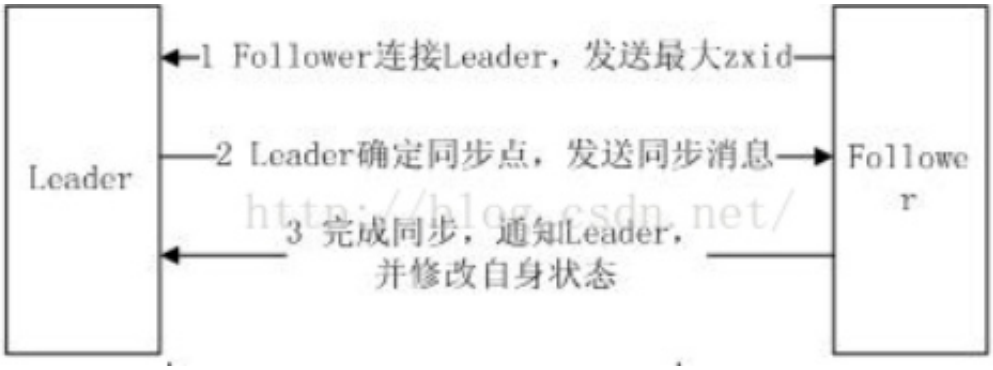
1、Leader 等待 server 连接;

2、Follower 连接 leader，将最大的 zxid 发送给 leader;

3、Leader 根据 follower 的 zxid 确定同步点;

4、完成同步后通知 follower 已经成为 uptodate 状态;

5、Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了。



19.分布式通知和协调

对于系统调度来说:操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户

端。
对于执行情况汇报:每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

20.机器中为什么会有 leader?
在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

21.zk 节点宕机如何处理?
Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。
如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失;
如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。
ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。
所以
3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)
2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

22.zookeeper 负载均衡和 nginx 负载均衡区别
zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件;但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

23.zookeeper watch 机制
Watch 机制官方声明:一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。
Zookeeper 机制的特点:
1、一次性触发数据发生改变时，一个 watcher event 会被发送到 client，但是 client 只会收到一次这样的信息。
2、watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，

由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了 变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。
3、数据监视 Zookeeper 有数据监视和子数据监视 getdata() and exists()设置数据监视，getchildren()设置了子节点监视。
4、注册 watcher getData、exists、getChildren
5、触发 watcher create、delete、setData
6、setData()会触发 znode 上设置的 data watch(如果 set 成功的话)。一个成功的 create() 操作会触发被创建的 znode 上的数据 watch，以及其父节点上的 child watch。而一个成功的 delete()操作将会同时触发一个 znode 的 data watch 和 child watch(因为这样就没有子节点了)，同时也会触发其父节点的 child watch。
7、当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失:对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。
8、Watch 是轻量级的，其实就是本地 JVM 的 Callback，服务器端只是存了是否有设置了 Watcher 的布尔类型

5.1.2 Nginx面试专题

1、请解释一下什么是 Nginx?
Nginx 是一个 web 服务器和反向代理服务器，用于 HTTP、HTTPS、SMTP、POP3 和 IMAP 协议。

2、请列举 Nginx 的一些特性。
Nginx 服务器的特性包括:
反向代理/L7 负载均衡器
嵌入式 Perl 解释器
动态二进制升级
可用于重新编写 URL，具有非常好的 PCRE 支持

3、请解释 Nginx 如何处理 HTTP 请求。

Nginx 使用反应器模式。主事件循环等待操作系统发出准备事件的信号，这样数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可以提供数万个并发连接。

4、在 Nginx 中，如何使用未定义的服务器名称来阻止处理请求?

只需将请求删除的服务器就可以定义为:

```
Server {listen 80;server_name “ “ ;return 444;
}
```

这里，服务器名被保留为一个空字符串，它将在没有“主机”头字段的情况下匹配请求，而一个特殊的 Nginx 的非标准代码 444 被返回，从而终止连接。

5、 使用“反向代理服务器”的优点是什么?

反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和 web 服务器之间的中间层。这对于安全方面来说是很好的，特别是当您使用 web 托管服务时。

6、请列举 Nginx 服务器的最佳用途。

Nginx 服务器的最佳用法是在网络上部署动态 HTTP 内容，使用 SCGI、WSGI 应用程序服务器、用于脚本的 FastCGI 处理程序。它还可以作为负载均衡器。

7、请解释 Nginx 服务器上的 Master 和 Worker 进程分别是什么?

Master 进程:读取及评估配置和维持

Worker 进程:处理请求

8、请解释你如何通过不同于 80 的端口开启 Nginx?

为了通过一个不同的端口开启 Nginx，你必须进入/etc/Nginx/sites-enabled/，如果这是默认文件，那么你必须打开名为“default”的文件。编辑文件，并放置在你想要的端口:

```
Like server { listen 81; }
```

9、请解释是否有可能将 Nginx 的错误替换为 502 错误、503?

502 =错误网关 503 =服务器超载

有可能，但是您可以确保 `fastcgi_intercept_errors` 被设置为 ON，并使用错误页面指令。

```
Location / {fastcgi_pass 127.0.01:9001;fastcgi_intercept_error
s on;error_page 502 =503/error_page.html;#...}
```

10、在 Nginx 中，解释如何在 URL 中保留双斜线?

要在 URL 中保留双斜线，就必须使用 `merge_slashes_off`；

语法: `merge_slashes [on/off]`

默认值: merge_slashes on

环境: http，server

11、请解释 `ngx_http_upstream_module` 的作用是什么?

`ngx_http_upstream_module` 用于定义可通过 fastcgi 传递、proxy 传递、uwsgi 传递、memcached 传递和 scgi 传递指令来引用的服务器组。

12、请解释什么是 C10K 问题?

C10K 问题是指无法同时处理大量客户端(10,000)的网络套接字。

13、请陈述 `stub_status` 和 sub_filter 指令的作用是什么?

`Stub_status` 指令:该指令用于了解 Nginx 当前状态的当前状态，如当前的活 动连接，接受和处理当前读/写/等待连接的总数

`Sub_filter` 指令:它用于搜索和替换响应中的内容，并快速修复陈旧的数据

14、解释 Nginx 是否支持将请求压缩到上游?

您可以使用 Nginx 模块 gunzip 将请求压缩到上游。gunzip 模块是一个过滤器，它可以对不支持“gzip”编码方法的客户机或服务器使用“内容编码:gzip”来解压缩响应。

15、解释如何在 Nginx 中获得当前的时间?

要获得 Nginx 的当前时间，必须使用 SSI 模块、`$date_gmt` 和 `$date_local` 的变量。

```
Proxy_set_header THE-TIME $date_gmt;
```

16、用 Nginx 服务器解释-s 的目的是什么?

用于运行 Nginx -s 参数的可执行文件。

17、解释如何在 Nginx 服务器上添加模块?

在编译过程中，必须选择 Nginx 模块，因为 Nginx 不支持模块的运行时间选择。

5.2、分布式通讯面试整理

5.2.1 RabbitMQ消息中间件面试专题

1、RabbitMQ 中的 broker 是指什么?cluster 又是指什么?

broker 是指一个或多个 erlang node 的逻辑分组，且 node 上运行着 RabbitMQ 应用程序。cluster 是在 broker 的基础之上，增加了 node 之间共享元数据的约束。

2、什么是元数据?元数据分为哪些类型?包括哪些内容?与 cluster 相关的元数据 有哪些?元数据是如何保存的?元数据在 cluster 中是如何分布的?

在非 cluster 模式下，元数据主要分为 Queue 元数据(queue 名字和属性等)、 Exchange 元数据(exchange 名字、类型和属性等)、Binding 元数据(存放路由关系的查找表)、Vhost 元数据(vhost 范围内针对前三者的名字空间约束和安全属性设置)。在 cluster 模式下，还包括 cluster 中 node 位置信息和 node 关系信息。元数据按照 erlang node 的类型确定是仅保存于 RAM 中，还是同时保存在 RAM 和 disk 上。元数据在 cluster 中是全 node 分布的。

3、RAM node 和 disk node 的区别?

RAM node 仅将 fabric(即 queue、exchange 和 binding等 RabbitMQ基础构件)相关元数据保存到内存中，但 disk node 会在内存和磁盘中均进行存储。RAM node 上唯一会存储到磁盘上的元数据是 cluster 中使用的 disk node 的地址。要求在 RabbitMQ cluster 中至少存在一个 disk node 。

4、RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制?

可以认为是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。

5、vhost 是什么?起什么作用?

vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离 的手段(一个典型的例子就是不同的应用可以跑在不同的 vhost 中)。

6、在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange ，以及进行 binding 会有什么不同?

当你在单 node 上声明 queue 时，只要该 node 上相关元数据进行了变

更，你就会得到 Queue.Declare-ok 回应;而在 cluster 上声明 queue ，则要求 cluster 上的全部 node 都要进行元数据成功更新，才会得到 Queue.Declare-ok 回应。另外，若 node 类型 为 RAM node 则变更的数据仅保存在内存中，若类型为 disk node 则还要变更保存在磁盘上的数据。

7、客户端连接到 cluster 中的任意 node 上是否都能正常工作?

是的。客户端感觉不到有何不同。

8、若 cluster 中拥有某个 queue 的 owner node 失效了，且该 queue 被声明具有 durable 属性，是否能够成功从其他 node 上重新声明该 queue ?

不能，在这种情况下，将得到 404 NOT_FOUND 错误。只能等 queue 所属的 node 恢复后才能使用该 queue 。但若该 queue 本身不具有 durable 属性，则可在其他 node 上重新声明。

9、cluster 中 node 的失效会对 consumer 产生什么影响?若是在 cluster 中创建了 mirrored queue ，这时 node 失效会对 consumer 产生什么影响?

若是 consumer 所连接的那个 node 失效(无论该 node 是否为 consumer 所订阅 queue 的 owner node)，则 consumer 会在发现 TCP 连接断开时，按标准行为执行重连逻辑，并根据“Assume Nothing”原则重建相应的 fabric 即可。若是失效的 node 为 consumer 订阅 queue 的owner node，则 consumer 只能通过 Consumer Cancellation Notification 机制来检测与该 queue 订阅关系的终止，否则会出现傻等却没有任何消息来 到的问题。

10、能够在地理上分开的不同数据中心使用 RabbitMQ cluster 么?

不能。

第一，你无法控制所创建的 queue 实际分布在 cluster 里的哪个 node 上(一般使用 HAProxy + cluster 模型时都是这样)，这可能会导致各种跨地域访问时的常见问题;

第二，Erlang 的 OTP 通信框架对延迟的容忍度有限，这可能会触发各种超时，导致业务疲于处理;

第三，在广域网上的连接失效问题将导致经典的“脑裂”问题，而 RabbitMQ 目前无法处理(该问题主要是说 Mnesia)。

11、为什么 heavy RPC 的使用场景下不建议采用 disk node

heavy RPC 是指在业务逻辑中高频调用 RabbitMQ 提供的 RPC 机制，导

致不断创建、 销毁 reply queue ， 进而造成 disk node 的性能问题(因为会针对元数据不断写盘)。所以在使用 RPC 机制时需要考虑自身的业务场景。

12、向不存在的 exchange 发 publish 消息会发生什么?向不存在的 queue 执行consume 动作会发生什么?

都会收到 Channel.Close 信令告之不存在(内含原因 404 NOT_FOUND)。

13、 `routing_key` 和 `binding_key` 的最大长度是多少?

255 字节。

14、RabbitMQ 允许发送的 message 最大可达多大?

根据 AMQP 协议规定，消息体的大小由 64-bit 的值来指定，所以你就可以知道到底能发多大的数据了。

15、什么情况下 producer 不主动创建 queue 是安全的?

1.message 是允许丢失的;
2.实现了针对未处理消息的 republish 功能(例如采用 Publisher Confirm 机制)。

16、“dead letter”queue 的用途?

当消息被 RabbitMQ server 投递到 consumer 后，但 consumer 却通过 Basic.Reject 进行了拒绝时(同时设置 requeue=false)，那么该消息会被放入“dead letter”queue 中。 该 queue 可用于排查 message 被 reject 或 undeliver 的原因。

17、什么说保证 message 被可靠持久化的条件是 queue 和 exchange 具有 durable 属性，同时 message 具有 persistent 属性才行?

binding 关系可以表示为 exchange – binding – queue 。从文档中我们知道，若要求投递的 message 能够不丢失，要求 message 本身设置 persistent 属性，要求 exchange 和 queue 都设置 durable 属性。其实这问题可以这么想，若 exchange 或 queue 未设置 durable 属性，则在其 crash 之后就会无法恢复，那么即使 message 设置了 persistent 属性，仍然存在 message 虽然能恢复但却无处容身的问题;同理，若 message 本身未设置 persistent 属性，则 message 的持久化更无从谈起。

18、什么情况下会出现 blackholed 问题?

blackholed 问题是指，向 exchange 投递了 message ， 而由于各种原因导致该 message 丢失，但发送者却不知道。可导致 blackholed 的情况:

1.向未绑定 queue 的 exchange 发送 message;
2.exchange 以 `binding_key key_A` 绑定了 `queue queue_A`，但向 该 exchange 发送 message 使用的 `routing_key` 却是 `key_B` 。

19、如何防止出现 blackholed 问题?

没有特别好的办法，只能在具体实践中通过各种方式保证相关 fabric 的存在。另外，如果在执行 Basic.Publish 时设置 mandatory=true ，则在遇到可能出现 blackholed 情况 时，服务器会通过返回 Basic.Return 告之当前 message 无法被正确投递(内含原因 312 NO_ROUTE)。

20、Consumer Cancellation Notification 机制用于什么场景?

用于保证当镜像 queue 中 master 挂掉时，连接到 slave 上的 consumer 可以收到自身 consume 被取消的通知，进而可以重新执行 consume 动作从新选出的 master 出获得消息。若不采用该机制，连接到 slave 上的 consumer 将不会感知 master 挂掉这个事情，导致后续无法再收到新 master 广播出来的 message 。另外，因为在镜像 queue 模式下，存在将 message 进行 requeue 的可能，所以实现 consumer 的逻辑时需要能够正确处理出现重复 message 的情况。

21、Basic.Reject 的用法是什么?

该信令可用于 consumer 对收到的 message 进行 reject 。若在该信令中设置 requeue=true，则当 RabbitMQ server 收到该拒绝信令后，会将该 message 重新发送到下一个处于 consume 状态的 consumer 处(理论上仍可能将该消息发送给当前 consumer)。若设置 requeue=false ，则 RabbitMQ server 在收到拒绝信令后，将直接将该 message 从 queue 中移除。

另外一种移除 queue 中 message 的小技巧是，consumer 回复 Basic.Ack 但不对获取到的 message 做任何处理。

而 Basic.Nack 是对 Basic.Reject 的扩展，以支持一次拒绝多条 message 的能力。

22、为什么不应该对所有的 message 都使用持久化机制?

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现“坑爹”问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该

queue 的 message 将被 blackholed ;若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。所 以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想 达到 100,000 条/秒以上的消息吞吐量(单 RabbitMQ 服务器)，则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化(例如使 用 SSD)。另外一种处理原则是:仅对关键消息作持久化处理(根据业务重要程度)，且应 该保证关键消息的量不会导致性能瓶颈。

23、RabbitMQ 中的 cluster、mirrored queue，以及 warrens 机制分别用于解决 什么问题?存在哪些问题?

cluster是为了解决当 cluster 中的任意 node 失效后，producer 和 consumer 均可以 通过其他 node 继续工作，即提高了可用性;另外可以通过增加 node 数量增加 cluster 的消息吞吐量的目的。cluster 本身不负责 message 的可靠性问题(该问题由 producer 通 过各种机制自行解决);cluster 无法解决跨数据中心的问题(即脑裂问题)。

另外，在 cluster 前使用 HAProxy 可以解决 node 的选择问题，即业务无需知道 cluster 中多个 node 的 ip 地址。可以利用 HAProxy 进行失效 node 的探测，可以作负载均衡。

Mirrored queue 是为了解决使用 cluster 时所创建的 queue 的完整信息仅存在于单一 node 上的问题，从另一个角度增加可用性。若想正确使用该功能，需要保证:

1.consumer 需要支持 Consumer Cancellation Notification 机制;

2.consumer 必须能够正确处理重复 message 。

Warrens是为了解决 cluster 中 message 可能被 blackholed 的问题，即不能接受 producer 不停 republish message 但 RabbitMQ server 无回应的情况。Warrens 有两种构成方式:

一种模型是两台独立的 RabbitMQ server + HAProxy ，其中两个 server 的状态分别为 active 和 hot-standby 。该模型的特点为:两台 server 之间无任何数据共享和协议交互，两台 server 可以基于不同的 RabbitMQ 版本。

另一种模型为两台共享存储的 RabbitMQ server + keepalived，其中两个 server 的状态分 别为 active 和 cold-standby。

该模型的特点为:两台 server 基于共享存储可以做到完全 恢复，要求必须基于完全相同的 RabbitMQ 版本。

Warrens 模型存在的问题:
对于第一种模型，虽然理论上讲不会丢失消息，但若在该模型上使用持久化机制，就会出现这样一种情况，即若作为 active 的 server 异常后，持久化 在该 server 上的消息将暂时无法被 consume ，因为此时该 queue 将无法在作为 hot- standby 的 server 上被重建，所以，只能等到异常的 active server 恢复后，才能从其上的 queue 中获取相应的 message 进行处理。
而对于业务来说，需要具有:a.感知 AMQP 连接断开后重建各种 fabric 的能力;b.感知 active server 恢复的能力;c.切换回 active server 的时机控制，以及切回后，针对 message 先后顺序产生的变化进行处理的能力。
对于第二种模型，因为是基于共享存储的模式，所以导致 active server 异常的条件，可能同样会导致 cold-standby server 异常;另外，在该模型下，要求 active 和 cold-standby 的 server 必须具有相同的 node 名和 UID ，否则将产生访问权限问题;最后，由于该模型是冷备方案，故无法保证 cold-standby server 能在你要求的时限内成功启动。

5.2.2 Kafka面试专题

1.Kafka 的设计时什么样的呢?
Kafka 将消息以 topic 为单位进行归纳
将向 Kafka topic 发布消息的程序成为 producers.
将预订 topics 并消费消息的程序成为 consumer.
Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker. producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息

2.数据传输的事物定义有哪三种?
数据传输的事务定义通常有以下三种级别:
(1)最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输
(2)最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.
(3)精确的一次(Exactly once): 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

3.Kafka 判断一个节点是否还活着有那两个条件?

(1)节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接
(2)如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久

4.producer 是否直接将数据发送到 broker 的 leader(主节点)?
producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点进行分发，为了帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知:哪些节点是活动的，目标 topic 目标分区的 leader 在哪。这样 producer 就可以直接将消息发送到目的地了

5、Kafa consumer 是否可以消费指定分区消息?
Kafa consumer 消费消息时，向 broker 发出"fetch"请求去消费特定分区的消息，consumer 指定消息在日志中的偏移量(offset)，就可以消费从这个位置开始的消息，customer 拥有 了 offset 的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的

6、Kafka 消息是采用 Pull 模式，还是 Push 模式?
Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计:producer 将消息推送到 broker，consumer 从 broker 拉取消息
一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式，将消息推送到下游的 consumer。
这样做有好处也有坏处:由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时， consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式
Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一 次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略
Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询， 直到新消息到 t 达。为了避免这点，Kafka 有个参数可

以让 consumer 阻塞知道新消息到达 (当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

7.Kafka 存储在硬盘上的消息格式是什么?

消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和 CRC32 校验码。

•消息长度: 4 bytes (value: 1+4+n)

•版本号: 1 byte

•CRC 校验码: 4 bytes

•具体的消息: n bytes

8.Kafka 高效文件存储设计特点:

(1).Kafka 把 topic 中一个 parition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

(2).通过索引信息可以快速定位 message 和确定 response 的最大大小。

(3).通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。

(4).通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

9.Kafka 与传统消息系统之间有三个关键区别

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统:它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

10.Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中

• 副本因子不能大于 Broker 的个数;

• 第一个分区(编号为 0)的第一个副本放置位置是随机从 brokerList 选择的;

• 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个 Broker，5个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上;第三个分区将会放在第一个 Broker 上;第四个分区将会放在第二个 Broker 上，依次类推;

• 剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是 随机产生的

11.Kafka 新建的分区会在哪个目录下创建

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数

据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。
当然我们也可以配置 log.dir 参数，含义一样。只需要设置其中一个即可。
如果 log.dirs 参数只配置了一个目录，那么分配到各个 Broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。
但是如果 log.dirs 参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢? 答案是:Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic 名+分区 ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录!也就是说，如果你给 log.dirs 参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

12.partition 的数据如何保存到硬盘

topic 中的多个 partition 以文件夹的形式保存到 broker，每个分区序号从 0 递增，且消息有序
Partition 文件下有多个 segment(xxx.index，xxx.log)
segment 文件里的大小和配置文件大小一致可以根据要求修改 默认为 1g 如果大小大于 1g 时，会滚动一个新的 segment 并且以上一个 segment 最后一条消息的偏移量命名

13.kafka 的 ack 机制

request.required.acks 有三个值 0 1 -1
0:生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候就会丢数据
1:服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他不确保是否复制完成新 leader 也会导致数据丢失
-1:同样在1的基础上服务端会等所有的follower的副本受到数据后才会受到 leader发出的 ack，这样数据不会丢失

14.Kafka 的消费者如何消费数据

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量(offset)的位置 等到下次消费时，他会接着上次位置继续消费

15.消费者负载均衡策略

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员

16.数据有序

一个消费者组里它的内部是有序的
消费者组与消费者组之间是无序的

17.kafaka 生产数据时数据的分组策略

生产者决定数据产生到集群的哪个 partition 中每一条消息都是以(key，value)格式
Key 是由生产者发送数据传入所以生产者(key)决定了数据产生到集群的哪个 partition

ActiveMQ消息中间件面试专题

1.什么是 ActiveMQ?

activeMQ 是一种开源的，实现了 JMS1.1 规范的，面向消息(MOM)的中间件，为应用程序提供高效的、可扩展的、稳定的和安全的企业级消息通信

2. ActiveMQ 服务器宕机怎么办?

这得从 ActiveMQ 的储存机制说起。在通常的情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的，它们的最大限制在配置文件的 `<systemUsage>` 节点中配置。但是，在非持久化消息堆 积到一定程度，内存告急的时候，ActiveMQ 会将内存中的非持久化消息写入临时文件中，以腾出内存。 虽然都保存到了文件里，但它和持久化消息的区别是，重启后持久化消息会从文件中恢复，非持久化的临 时文件会直接删除。
那如果文件增大到达了配置中的最大限制的时候会发生什么?我做了以下实验:
设置 2G 左右的持久化文件限制，大量生产持久化消息直到文件达到最大限制，此时生产者阻塞，但消费者可正常连接并消费消息，等消息消费掉一部分，文件删除又腾出空间之后，生产者又可继续发送消息，服务自动恢复正常。
设置 2G 左右的临时文件限制，大量生产非持久化消息并写入临时文件，在达到最大限制时，生产者阻塞，消费者可正常连接但不能消费消息，或者原本慢速消费的消费者，消费突然停止。整个系统可连接， 但是无法提供服务，就这样挂了。
具体原因不详，解决方案:尽量不要用非持久化消息，非要用的话，将临时文件限制尽可能的调大。

3. 丢消息怎么办?

这得从 java 的 java.net.SocketException 异常说起。简单点说就是当网络

发送方发送一堆数据，然后调用 close 关闭连接之后。这些发送的数据都在接收者的缓存里，接收者如果调用 read 方法仍旧能从缓存中读取这些数据，尽管对方已经关闭了连接。但是当接收者尝试发送数据时，由于此时连接已关闭，所以会发生异常，这个很好理解。不过需要注意的是，当发生 SocketException 后，原本缓存区中数据也作废了，此时接收者再次调用 read 方法去读取缓存中的数据，就会报 Software caused connection abort: recv failed 错误。
通过抓包得知，ActiveMQ 会每隔 10 秒发送一个心跳包，这个心跳包是服务器发送给客户端的，用来判断客户端死没死。如果你看过上面第一条，就会知道非持久化消息堆积到一定程度会写到文件里，这个写 的过程会阻塞所有动作，而且会持续 20 到 30 秒，并且随着内存的增大而增大。当客户端发完消息调用 connection.close()时，会期待服务器对于关闭连接的回答，如果超过 15 秒没回答就直接调用 socket 层的 close 关闭 tcp 连接了。这时客户端发出的消息其实还在服务器的缓存里等待处理，不过由于服务器心跳包的设置，导致发生了 java.net.SocketException 异常，把缓存里的数据作废了，没处理的消息全部丢失。
解决方案:用持久化消息，或者非持久化消息及时处理不要堆积，或者启动事务，启动事务后，commit()方法会负责任的等待服务器的返回，也就不会关闭连接导致消息丢失了。

4. 持久化消息非常慢。
默认的情况下，非持久化的消息是异步发送的，持久化的消息是同步发送的，遇到慢一点的硬盘，发送消息的速度是无法忍受的。但是在开启事务的情况下，消息都是异步发送的，效率会有 2 个数量级的提升。 所以在发送持久化消息时，请务必开启事务模式。其实发送非持久化消息时也建议开启事务，因为根本不会影响性能。

5. 消息的不均匀消费。
有时在发送一些消息之后，开启 2 个消费者去处理消息。会发现一个消费者处理了所有的消息，另一个消费者根本没收到消息。原因在于 ActiveMQ 的 prefetch 机制。当消费者去获取消息时，不会一条一条去获取，而是一次性获取一批，默认是 1000 条。这些预获取的消息，在还没确认消费之前，在管理控制台还是可以看见这些消息的，但是不会再分配给其他消费者，此时这些消息的状态应该算作“已分配未消 费”，如果消息最后被消费，则会在服务器端被删除，如果消费者崩溃，则这些消息会被重新分配给新的消费者。但是如果消费者既不消费确认，又不崩溃，那这

些消息就永远躺在消费者的缓存区里无法处理。更通常的情况是，消费这些消息非常耗时，你开了 10 个消费者去处理，结果发现只有一台机器吭哧吭哧处理，另外 9 台啥事不干。
解决方案:将 prefetch 设为 1，每次处理 1 条消息，处理完再去取，这样也慢不了多少。

6. 死信队列。
如果你想在消息处理失败后，不被服务器删除，还能被其他消费者处理或重试，可以关闭 `AUTO_ACKNOWLEDGE`，将 ack 交由程序自己处理。那如果使用了 `AUTO_ACKNOWLEDGE`，消息是什么时候被确认的，还有没有阻止消息确认的方法?有!
消费消息有 2 种方法，一种是调用 consumer.receive()方法，该方法将阻塞直到获得并返回一条消息。这种情况下，消息返回给方法调用者之后就自动被确认了。另一种方法是采用 listener 回调函数，在有消息到达时，会调用 listener 接口的 onMessage 方法。在这种情况下，在 onMessage 方法执行完毕后，消息才会被确认，此时只要在方法中抛出异常，该消息就不会被确认。那么问题来了，如果一条消息不能 被处理，会被退回服务器重新分配，如果只有一个消费者，该消息又会重新被获取，重新抛异常。就算有多个消费者，往往在一个服务器上不能处理的消息，在另外的服务器上依然不能被处理。难道就这么退回 --获取--报错死循环了吗?
在重试 6 次后，ActiveMQ 认为这条消息是“有毒”的，将会把消息丢到死信队列里。如果你的消息不见 了，去 ActiveMQ.DLQ 里找找，说不定就躺在那里。

7. ActiveMQ 中的消息重发时间间隔和重发次数吗?
ActiveMQ:是 Apache 出品，最流行的，能力强劲的开源消息总线。是一个完全支持 JMS1.1 和 J2EE 1.4 规范的 JMS Provider 实现。JMS(Java 消息服务):是一个 Java 平台中关于面向消息中间件 (MOM)的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。
首先，我们得大概了解下，在哪些情况下，ActiveMQ 服务器会将消息重发给消费者，这里为简单起见，假定采用的消息发送模式为队列(即消息发送者和消息接收者)。
1 如果消息接收者在处理完一条消息的处理过程后没有对 MOM 进行应答，则该消息将由 MOM 重发.
2 如果我们队某个队列设置了预读参数(consumer.prefetchSize)，如果消息接收者在处理第一条消息时(没向 MOM 发送消息接收确认)就宕机了，则

预读数量的所有消息都将被重发!
3 如果 Session 是事务的，则只要消息接收者有一条消息没有确认，或发送消息期间 MOM 或客户端某一方突然宕机了，则该事务范围中的所有消息 MOM 都将重发。
4 说到这里，大家可能会有疑问，ActiveMQ 消息服务器怎么知道消费者客户端到底是消息正在处理中 还没来得急对消息进行应答还是已经处理完成了没有应答或是宕机了根本没机会应答呢?其实在所有的客户端机器上，内存中都运行着一套客户端的 ActiveMQ 环境，该环境负责缓存发来的消息，负责维持着 和 ActiveMQ 服务器的消息通讯，负责失效转移(fail-over)等，所有的判断和处理都是由这套客户端环境来完成的。
我们可以来对 ActiveMQ 的重发策略(Redelivery Policy)来进行自定义配置，其中的配置参数主要有以下几个:
可用的属性
属性默认值说明
l collisionAvoidanceFactor 默认值 0.15 , 设置防止冲突范围的正负百分比，只有启用 useCollisionAvoidance 参数时才生效。
l maximumRedeliveries 默认值 6 , 最大重传次数，达到最大重连次数后抛出异常。为-1 时不限制次数，为 0 时表示不进行重传。
l maximumRedeliveryDelay 默认值-1, 最大传送延迟，只在 useExponentialBackOff 为 true 时有效 (V5.5)，假设首次重连间隔为 10ms，倍数为 2，那么第二次重连时间间隔为 20ms，第三次重连时间间隔为 40ms，当重连时间间隔大的最大重连时间间隔时，以后每次重连时间间隔都为最大重连时间间隔。
l initialRedeliveryDelay 默认值 1000L, 初始重发延迟时间
l redeliveryDelay 默认值 1000L, 重发延迟时间，当 initialRedeliveryDelay=0 时生效(v5.4)
l useCollisionAvoidance 默认值 false, 启用防止冲突功能，因为消息接收时是可以使用多线程并发处理的，应该是为了重发的安全性，避开所有并发线程都在同一个时间点进行消息接收处理。所有线程在同一个时间点处理时会发生什么问题呢?应该没有问题，只是为了平衡 broker 处理性能，不会有时很忙，有时很空闲。
l useExponentialBackOff 默认值 false, 启用指数倍数递增的方式增加延迟时间。
l backOffMultiplier 默认值 5, 重连时间间隔递增倍数，只有值大于 1 和启用 useExponentialBackOff 参数时才生效。

5.3、分布式数据库面试整理

5.3.1 redis面试专题

1、redis 和 memcached 什么区别?为什么高并发下有时单线程的 redis 比多线程的 memcached 效率要高?

区别:
1.mc 可缓存图片和视频。rd 支持除 k/v 更多的数据结构;
2.rd 可以使用虚拟内存，rd 可持久化和 aof 灾难恢复，rd 通过主从支持数据备份; 3.rd 可以做消息队列。
原因:mc 多线程模型引入了缓存一致性和锁，加锁带来了性能损耗。

2、redis 主从复制如何实现的?redis 的集群模式如何实现?redis 的 key 是如何寻址的?

主从复制实现:主节点将自己内存中的数据做一份快照，将快照发给从节点，从节点将数 据恢复到内存中。之后再每次增加新数据的时候，主节点以类似于 mysql 的二进制日志方 式将语句发送给从节点，从节点拿到主节点发送过来的语句进行重放。
分片方式:
-客户端分片
-基于代理的分片
- Twemproxy
- codis

-路由查询分片
- Redis-cluster(本身提供了自动将数据分散到 Redis Cluster 不同节点的能力，整个数据集合的某个数据子集存储在哪个节点对于用户来说是透明的)

redis-cluster 分片原理:Cluster 中有一个 16384 长度的槽(虚拟槽)，编号分别为 0-16383。 每个 Master 节点都会负责一部分的槽，当有某个 key 被映射到某个 Master 负责的槽，那么这个 Master 负责为这个 key 提供服务，至于哪个 Master 节点负责哪个槽，可以由用户指定，也可以在初始化的时候自动生成，只有 Master 才拥有槽的所有权。Master 节点维护着一个 16384/8 字节的位序列，Master 节点用 bit 来标识对于某个槽自己是否拥有。比如对于编号为 1 的槽，Master 只要判断序列的第二位(索引从 0 开始)是不是为 1 即可。这种结构很容易添加或者删除节点。比如如果我想新添加个节点 D, 我需要从节点 A、B、 C 中得部分槽到 D 上。

3、使用 redis 如何设计分布式锁?说一下实现思路?使用 zk 可以吗?如何

实现?这两种有什么区别?

redis:

1.线程 A setnx(上锁的对象,超时时的时间戳 t1)，如果返回 true，获得锁。

2.线程 B 用 get 获取 t1,与当前时间戳比较,判断是是否超时,没超时 false,若超时执行第 3 步;

3.计算新的超时时间 t2,使用 getset 命令返回 t3(该值可能其他线程已经修改过),如果 t1==t3，获得锁，如果 t1!=t3 说明锁被其他线程获取了。

4.获取锁后，处理完业务逻辑，再去判断锁是否超时，如果没超时删除锁，如果已超时，不用处理(防止删除其他线程的锁)。

zk:

1.客户端对某个方法加锁时，在 zk 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点 node1;

2.客户端获取该路径下所有已经创建的子节点，如果发现自己创建的 node1 的序号是最小的，就认为这个客户端获得了锁。

3.如果发现 node1 不是最小的，则监听比自己创建节点序号小的最大的节点，进入等待。

4.获取锁后，处理完逻辑，删除自己创建的 node1 即可。区别:zk 性能差一些，开销大，实现简单。

4、知道 redis 的持久化吗?底层如何实现的?有什么优点缺点?

RDB(Redis DataBase:在不同的时间点将 redis 的数据生成的快照同步到磁盘等介质上):内存到硬盘的快照，定期更新。缺点:耗时，耗性能(fork+io 操作)，易丢失数据。

AOF(Append Only File:将redis所执行过的所有指令都记录下来，在下次redis重启时，只需要执行指令就可以了):写日志。缺点:体积大，恢复速度慢。

bgsave 做镜像全量持久化，aof 做增量持久化。因为 bgsave 会消耗比较长的时间，不够实时，在停机的时候会导致大量的数据丢失，需要 aof 来配合，在 redis 实例重启时，优先使用 aof 来恢复内存的状态，如果没有 aof 日志，就会使用 rdb 文件来恢复。Redis 会定期做 aof 重写，压缩 aof 文件日志大小。Redis4.0 之后有了混合持久化的功能，将 bgsave 的全量和 aof 的增量做了融合处理，这样既保证了恢复的效率又兼顾了数据的安全性。bgsave 的原理，fork 和 cow, fork 是指 redis 通过创建子进程来进行 bgsave 操作，cow 指的是 copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分

离开来。

5、redis 过期策略都有哪些?LRU 算法知道吗?写一下 java 代码实现?

过期策略:

定时过期(一 key 一定时器)，惰性过期:只有使用 key 时才判断 key 是否已过期，过期则清除。定期过期:前两者折中。

```
LRU:new LinkedHashMap<K, V>(capacity, DEFAULT_LOAD_FACTORY, tr
ue);
//第三个参数置为 true，代表 linkedlist 按访问顺序排序，可作为 LRU 缓存
;设为 false 代表 按插入顺序排序，可作为 FIFO 缓存
```

LRU 算法实现:

1.通过双向链表来实现，新数据插入到链表头部;

2.每当缓存命中(即缓存 数据被访问)，则将数据移到链表头部;

3.当链表满的时候，将链表尾部的数据丢弃。

LinkedHashMap:HashMap 和双向链表合二为一即是 LinkedHashMap。HashMap 是无序 的，LinkedHashMap 通过维护一个额外的双向链表保证了迭代顺序。该迭代顺序可以是插入顺序(默认)，也可以是访问顺序。

6、缓存穿透、缓存击穿、缓存雪崩解决方案?

缓存穿透:指查询一个一定不存在的数据，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到 DB 去查询，可能导致 DB 挂掉。

解决方案:

1.查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短;

2.布 隆过滤器:将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对 DB 的查询。

缓存击穿:对于设置了过期时间的 key，缓存在某个时间点过期的时候，恰好这时间点对这个 Key 有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把 DB 压垮。

解决方案:

1.使用互斥锁:当缓存失效时，不立即去load db，先使用如Redis的setnx去设置一个互斥锁，当操作成功返回时再进行load db的操作并回设缓存，否则重试get缓存的方法。

2.永远不过期:物理不过期，但逻辑过期(后台异步线程去刷新)。缓存雪崩:设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。与缓存击穿的区别:雪崩是很多 key，击穿是某一个 key 缓存。
解决方案:将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值， 比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

7、在选择缓存时，什么时候选择 redis，什么时候选择 memcached
选择 redis 的情况:
1、复杂数据结构，value 的数据是哈希，列表，集合，有序集合等这种情况下，会选择 redis, 因为 memcache 无法满足这些数据结构，最典型的的使用场景是，用户订单列表，用户消息，帖子评论等。
2、需要进行数据的持久化功能，但是注意，不要把 redis 当成数据库使用，如果 redis 挂了，内存能够快速恢复热数据，不会将压力瞬间压在数据库上，没有 cache 预热的过程。对于只读和数据一致性要求不高的场景可以采用持久化存储
3、高可用，redis 支持集群，可以实现主动复制，读写分离，而对于 memcache 如果想要实现高可用，需要进行二次开发。
4、存储的内容比较大，memcache 存储的 value 最大为 1M。

选择 memcache 的场景:
1、纯 KV,数据量非常大的业务，使用 memcache 更合适，原因是，
a)memcache 的内存分配采用的是预分配内存池的管理方式，能够省去内存分配的时间，redis 是临时申请空间，可能导致碎片化。
b)虚拟内存使用，memcache 将所有的数据存储在物理内存里，redis 有自己的 vm 机制，理论上能够存储比物理内存更多的数据，当数据超量时，引发 swap,把冷数据刷新到磁盘上，从这点上，数据量大时，memcache 更快
c)网络模型，memcache 使用非阻塞的 IO 复用模型，redis 也是使用非阻塞的 IO 复用 模型，但是 redis 还提供了一些非 KV 存储之外的排序，聚合功能，复杂的 CPU 计算，会阻 塞整个 IO 调度，从这点上由于 redis 提供的功能较多，memcache 更快些
d) 线程模型，memcache使用多线程，主线程监听，worker子线程接受请求，执行读写，这个过程可能存在锁冲突。redis 使用的单线程，虽然无锁冲突，但是难以利用多核的特性提升吞吐量。

8、缓存与数据库不一致怎么办

假设采用的主存分离，读写分离的数据库，

如果一个线程 A 先删除缓存数据，然后将数据写入到主库当中，这个时候，主库和从库同 步没有完成，线程 B 从缓存当中读取数据失败，从从库当中读取到旧数据，然后更新至缓存，这个时候，缓存当中的就是旧的数据。

发生上述不一致的原因在于，主从库数据不一致问题，加入了缓存之后，主从不一致的时间被拉长了

处理思路:在从库有数据更新之后，将缓存当中的数据也同时进行更新，即当从库发生了数据更新之后，向缓存发出删除，淘汰这段时间写入的旧数据。

9、主从数据库不一致如何解决

场景描述，对于主从库，读写分离，如果主从库更新同步有时差，就会导致主从库数据的不一致

1、忽略这个数据不一致，在数据一致性要求不高的业务下，未必需要时时一致性

2、强制读主库，使用一个高可用的主库，数据库读写都在主库，添加一个缓存，提升数据读取的性能。

3、选择性读主库，添加一个缓存，用来记录必须读主库的数据，将哪个库，哪个表，哪个主键，作为缓存的 key,设置缓存失效的时间为主从库同步的时间，如果缓存当中有这个数据，直接读取主库，如果缓存当中没有这个主键，就到对应的从库中读取。

10、Redis 常见的性能问题和解决方案

1、master 最好不要做持久化工作，如 RDB 内存快照和 AOF 日志文件

2、如果数据比较重要，某个 slave 开启 AOF 备份，策略设置成每秒同步一次

3、为了主从复制的速度和连接的稳定性，master 和 Slave 最好在一个局域网内

4、尽量避免在压力大得主库上增加从库

5、主从复制不要采用网状结构，尽量是线性结构，Master<--Slave1<----Slave2 ....

11、Redis 的数据淘汰策略有哪些

voltile-lru 从已经设置过期时间的数据集中挑选最近最少使用的数据淘汰

voltile-ttl 从已经设置过期时间的数据库集当中挑选将要过期的数据

voltile-random 从已经设置过期时间的数据集任意选择淘汰数据 allkeys-lru 从数据集中挑选最近最少使用的数据淘汰
allkeys-random 从数据集中任意选择淘汰的数据 no-eviction 禁止驱逐数据

12、Redis 当中有哪些数据结构

字符串 String、字典 Hash、列表 List、集合 Set、有序集合 SortedSet。
如果是高级用户，那么还会有，如果你是 Redis 中高级用户，还需要加上下面几种数据结构 HyperLogLog、 Geo、Pub/Sub。

13、假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来?

使用 keys 指令可以扫出指定模式的 key 列表。
对方接着追问:如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题?
这个时候你要回答 redis 关键的一个特性:redis 的单线程的。keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 scan 指 令，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客 户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

14、使用 Redis 做过异步队列吗，是如何实现的

使用 list 类型保存数据信息，rpush 生产消息，lpop 消费消息，当 lpop 没有消息时，可以 sleep 一段时间，然后再检查有没有信息，如果不想 sleep 的话，可以使用 blpop, 在没 有信息的时候，会一直阻塞，直到信息的到来。redis 可以通过 pub/sub 主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

15、Redis 如何实现延时队列

使用 sortedset，使用时间戳做 score, 消息内容作为 key,调用 zadd 来生产消息，消费者使用 zrangbyscore 获取 n 秒之前的数据做轮询处理。

16、什么是 Redis?简述它的优缺点?

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。
因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。
Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据

结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能。
比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性 能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。
另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一个功能加强版的memcac hed 来用。 Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

17、Redis 相比 memcached 有哪些优势?
(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
(2) redis的速度比memcached快很多
(3) redis可以持久化其数据

18、Redis 支持哪几种数据类型?
String、List、Set、Sorted Set、hashes

19、Redis 主要消耗什么物理资源?
内存。

20、Redis 的全称是什么?
Remote Dictionary Server。

21、Redis 有哪几种数据淘汰策略?
noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令(大部分的写入指令，但 DEL 和几个例外)
allkeys-lru: 尝试回收最少使用的键(LRU)，使得新添加的数据有空间存放。
volatile-lru: 尝试回收最少使用的键(LRU)，但仅限于在过期集合的键,使得新添加的数据有空间存放。
allkeys-random: 回收随机的键使得新添加的数据有空间存放。
volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
volatile-ttl: 回收在过期集合的键，并且优先回收存活时间(TTL)较短的键,使得新添加的数据有空间存放。

22、Redis 官方为什么不提供 Windows 版本?
因为目前 Linux 版本已经相当稳定，而且用户量很大，无需开发 windows 版本，反而会带来兼容性等问题。

23、一个字符串类型的值能存储最大容量是多少?

512M

24、为什么 Redis 需要把所有数据放到内存中?

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。

所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。

在内存越来越便宜的今天，redis将会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

25、Redis 集群方案应该怎么做?都有哪些方案?

1.codis。

目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新 hash 节点。

2.redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

3.在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对key 进行hash计算，然后去对应的 redis 实例操作数据。这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方 案，数据震荡后的自动脚本恢复，实例的监控，等等。

26、Redis 集群方案什么情况下会导致整个集群不可用?

有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了，那么整个集群就会以为缺少 5 501-11000 这个范围的槽而不可用。

27、MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据?

redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

28、Redis 有哪些适合的场景?

(1)会话缓存(Session Cache)

最常用的一种使用Redis的情景是会话缓存(session cache)。用Redis缓存会话比其他存储(如Mem cached)的优势在于:Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车 信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗?

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis

来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。

(2)全页缓存(FPC)

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实 例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。

此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快

速度加载你曾浏览过的页面。

(3)队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言(如Python)对 list 的 push/pop 操作。

如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是 利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作 为 broker，你可以从这里去查看。

(4)排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合(Set)和有序集合(Sorted Set)也

使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。 所以，我们要从排序集合中获取到排名最靠前的 10 个用户–我们称之为 `“user_scores”` ，我们只需要像下

面一样执行即可:

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行:

```
ZRANGE user_scores 0 10 WITHSCORES
```

Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

(5)发布/订阅

最后(但肯定不是最不重要的)是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发

布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建 立聊天系统!

29、Redis 支持的 Java 客户端都有哪些?官方推荐用哪个?

Redisson、Jedis、lettuce 等等，官方推荐使用 Redisson。

30、Redis 和 Redisson 有什么关系?

Redisson 是一个高级的分布式协调 Redis 客服端，能帮助用户在分布式环境中轻松实现一些 Java 的对象 (Bloom filter, BitSet, Set, SetMultimap, ScoredSortedSet, SortedSet, Map, ConcurrentMap, List, List Multimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, ReadWriteLock, Atomi cLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

31、Jedis 与 Redisson 对比有什么优缺点?

Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持;

Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

32、Redis 如何设置密码及验证密码?

设置密码:config set requirepass 123456

授权密码:auth 123456

33、说说 Redis 哈希槽的概念?

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

34、Redis 集群的主从复制模型是怎样的?

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有 N-1 个复制品.

35、Redis 集群会有写操作丢失吗?为什么?

Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

36、Redis 集群之间是如何复制的?

异步复制

37、Redis 集群最大节点个数是多少?

16384 个。

38、Redis 集群如何选择数据库?

Redis 集群目前无法做数据库选择，默认在 0 数据库。

39、怎么测试 Redis 的连通性?

ping

40、Redis 中的管道有什么用?

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道(pipelining)，是一种几十年来广泛使用的技术。例如许多 POP3 协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。

41、怎么理解 Redis 事务?

事务是一个单独的隔离操作:事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作:事务中的命令要么全部被执行，要么全部都不执行。

42、Redis 事务相关的命令有哪几个?

MULTI、EXEC、DISCARD、WATCH

43、Redis key的过期时间和永久有效分别怎么设置?

EXPIRE 和 PERSIST 命令。

44、Redis 如何做内存优化?

尽可能使用散列表(hashes)，散列表(是说散列表里面存储的数少)使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。

45、Redis 回收进程如何工作的?

一个客户端运行了新的命令，添加了新的数据。

Redi 检查内存使用情况，如果大于 maxmemory 的限制, 则根据设定好的策略进行回收。

一个新的命令被执行，等等。
所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。
如果一个命令的结果导致大量内存被使用(例如很大的集合的交集保存到一个新的键)，不用多久内存限制就会被这个内存使用量超越。

5.3.2 MongoDB面试专题

1. 你说的 NoSQL 数据库是什么意思?NoSQL 与 RDBMS 直接有什么区别?为什么要使用和不使用 NoSQL 数据库?说一说 NoSQL 数据库的几个优点?

NoSQL 是非关系型数据库，NoSQL = Not Only SQL。
关系型数据库采用的结构化的数据，NoSQL 采用的是键值对的方式存储数据。
在处理非结构化/半结构化的大数据时;在水平方向上进行扩展时;随时应对动态增加的数据项时可以优 先考虑使用 NoSQL 数据库。
在考虑数据库的成熟度;支持;分析和商业智能;管理及专业性等问题时，应优先考虑关系型数据库。

2. NoSQL 数据库有哪些类型?

例如:MongoDB, Cassandra, CouchDB, Hypertable, Redis, Riak, Neo4j, HBASE, Couchbase, MemcacheDB, RevenDB and Voldemort are the examples of NoSQL databases.

3. MySQL 与 MongoDB 之间最基本的差别是什么?

MySQL 和 MongoDB 两者都是免费开源的数据库。MySQL 和 MongoDB 有许多基本差别包括数据的表示(data representation)，查询，关系，事务，schema 的设计和定义，标准化(normalization)，速度和性能。
通过比较 MySQL 和 MongoDB，实际上我们是在比较关系型和非关系型数据库，即数据存储结构不同。

4. 你怎么比较 MongoDB、CouchDB 及 CouchBase?

MongoDB 和 CouchDB 都是面向文档的数据库。MongoDB 和 CouchDB 都是开源 NoSQL 数据库的最典型代表。 除了都以文档形式存储外它们没有其他的共同点。MongoDB 和 CouchDB 在数据模型实现、接口、对象存储以及复制方法等方面有很多不同。

5. MongoDB 成为最好 NoSQL 数据库的原因是什么?

以下特点使得 MongoDB 成为最好的 NoSQL 数据库:

- 面向文件的
- 高性能
- 高可用性
- 易扩展性
- 丰富的查询语言

6. 32 位系统上有什么细微差别?

journaling 会激活额外的内存映射文件。这将进一步抑制 32 位版本上的数据库大小。因此，现在 journaling 在 32 位系统上默认是禁用的。

7. journal 回放在条目(entry)不完整时(比如恰巧有一个中途故障了)会遇到问题吗?

每个 journal (group)的写操作都是一致的，除非它是完整的否则在恢复过程中它不会回放。

8. 分析器在 MongoDB 中的作用是什么?

MongoDB 中包括了一个可以显示数据库中每个操作性能特点的数据库分析器。通过这个分析器你可以找到比预期慢的查询(或写操作);利用这一信息，比如，可以确定是否需要添加索引。

9. 名字空间(namespace)是什么?

MongoDB 存储 BSON 对象在丛集(collection)中。数据库名字和丛集名字以句点连结起来叫做名字空间 (namespace)。

10. 如果用户移除对象的属性，该属性是否从存储层中删除?

是的，用户移除属性然后对象会重新保存(re-save())。

11. 能否使用日志特征进行安全备份?

是的。

12.允许空值 null 吗?

对于对象成员而言，是的。然而用户不能够添加空值(null)到数据库丛集(collection)因为空值不是对象。 然而用户能够添加空对象{}。

13. 更新操作立刻 fsync 到磁盘?

不会，磁盘写操作默认是延迟执行的。写操作可能在两三秒(默认在 60 秒内)后到达磁盘。例如，如果一秒内数据库收到一千个对一个对象递增的操作，仅刷新磁盘一次。(注意，尽管 fsync 选项在命令行和经 过 getLastError_old 是有效的)(译者:也许是坑人的面试题??)。

14. 如何执行事务/加锁?

MongoDB 没有使用传统的锁或者复杂的带回滚的事务，因为它设计的宗旨是轻量，快速以及可预计的高性能。可以把它类比成 MySQL MyISAM 的自动提交模式。通过精简对事务的支持，性能得到了提升，特别是在一个可能会穿过多个服务器的系统里。

15. 为什么我的数据文件如此庞大?

MongoDB 会积极的预分配预留空间来防止文件系统碎片。

16. 启用备份故障恢复需要多久?

从备份数据库声明主数据库宕机到选出一个备份数据库作为新的主数据库将花费 10 到 30 秒时间。这期间在主数据库上的操作将会失败--包括写入和强一致性读取(strong consistent read)操作。然而，你还 能在第二数据库上执行最终一致性查询(eventually consistent query)(在 slaveOk 模式下)，即使在这段时间里。

17. 什么是 master 或 primary?

它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员。在一个备份集群中，当失效备援(failover)事件发生时，一个另外的成员会变成 primary。

18. 什么是 secondary 或 slave?

Seconday 从当前的 primary 上复制相应的操作。它是通过跟踪复制 oplog(local.oplog.rs)做到的。

19. 我必须调用 getLastError 来确保写操作生效了么?

不用。不管你有没有调用 getLastError(又叫"Safe Mode")服务器做的操作都一样。调用 getLastError 只 是为了确认写操作成功提交了。当然，你经常想得到确认，但是写操作的安全性和是否生效不是由这个决定的。

20. 我应该启动一个集群分片(sharded)还是一个非集群分片的 MongoDB 环境?

为开发便捷起见，我们建议以非集群分片(unsharded)方式开始一个 MongoDB 环境，除非一台服务器不足以存放你的初始数据集。从非集群分片升级到集群分片(sharding)是无缝的，所以在你的数据集还不 是很大的时候没必要考虑集群分片(sharding)。

21. 分片(sharding)和复制(replication)是怎样工作的?

每一个分片(shard)是一个分区数据的逻辑集合。分片可能由单一服务器或

者集群组成，我们推荐为每一个分片(shard)使用集群。

22. 数据在什么时候才会扩展到多个分片(shard)里?

MongoDB 分片是基于区域(range)的。所以一个集合(collection)中的所有的对象都被存放到一个块 (chunk)中。只有当存在多余一个块的时候，才会有多个分片获取数据的选项。现在，每个默认块的大小 是 64Mb，所以你需要至少 64 Mb 空间才可以实施一个迁移。

23. 当我试图更新一个正在被迁移的块(chunk)上的文档时会发生什么?

更新操作会立即发生在旧的分片(shard)上，然后更改才会在所有权转移(ownership transfers)前复制到新的分片上。

24. 如果在一个分片(shard)停止或者很慢的时候，我发起一个查询会怎样?

如果一个分片(shard)停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片(shard)响应很慢，MongoDB 则会等待它的响应。

25. 我可以把 moveChunk 目录里的旧文件删除吗?

没问题，这些文件是在分片(shard)进行均衡操作(balancing)的时候产生的临时文件。一旦这些操作已经 完成，相关的临时文件也应该被删除掉。但目前清理工作是需要手动的，所以请小心地考虑再释放这些文件的空间。

26. 我怎么查看 Mongo 正在使用的链接?

```
db._adminCommand("connPoolStats");
```

27. 如果块移动操作(moveChunk)失败了，我需要手动清除部分转移的文档吗?

不需要，移动操作是一致(consistent)并且是确定性的(deterministic);一次失败后，移动操作会不断重试; 当完成后，数据只会出现在新的分片里(shard)。

28. 如果我在使用复制技术(replication)，可以一部分使用日志(journaling)而其他部分则不使用吗?

可以。

29.当更新一个正在被迁移的块(Chunk)上的文档时会发生什么?

更新操作会立即发生在旧的块(Chunk)上，然后更改才会在所有权转移前复制到新的分片上。

30.MongoDB 在 A:{B,C}上建立索引，查询 A:{B,C}和 A:{C,B}都会使用索引吗?

不会，只会在 A:{B,C}上使用索引。

31.如果一个分片(Shard)停止或很慢的时候，发起一个查询会怎样?

如果一个分片停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片响应很慢，MongoDB 会等待它的响应。

32. MongoDB 支持存储过程吗?如果支持的话，怎么用?

MongoDB 支持存储过程，它是 javascript 写的，保存在 db.system.js 表中。

33.如何理解 MongoDB 中的 GridFS 机制，MongoDB 为何使用 GridFS 来存储文件?

GridFS 是一种将大型文件存储在 MongoDB 中的文件规范。使用 GridFS 可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了 BSON 对象有限制的问题。

5.3.3 memcached面试专题

1、memcached是怎么工作的?

Memcached的神奇来自两阶段哈希(two-stagehash)。Memcached就像一个巨大的、存储了很多 `<key,value>` 对的哈希表。通过key，可以存储或查询任意的数据。

客户端可以把数据存储在多台 memcached 上。当查询数据时，客户端首先参考节点列表计算出 key 的哈希值(阶段一哈希)，进而选中一个节点;客户端将请求发送给选中的节点，然后 memcached 节点通过一个内部的哈希算法(阶段二哈希)，查找真正的数据(item)。

举个列子，假设有3 个客服端 1 2 3 台 memcached A,B,C

Client 1 想把数据”barbaz”以key “foo”存储。Client 1 首先参考节点列表（A，B，C）计算 key “foo”的哈希值，假设 memcached B 被选中。接着，Client 1 直接 connect 到memcached B 通过 key “foo”把数据”barbaz”存储进去。 Client 2 使用与 Client 1 相同的客户端库(意味着阶段一的哈希算法相同)，也拥有同样的 memcached 列表(A，B，C）。于是，经过相同的哈希计算(阶段一)，Client 2 计算出 key “foo”在 memcached B 上，然后它直接请求 memcached B，得到数据”barbaz”。

各种客户端在 memcached 中数据的存储形式是不同的(perl Storable php serialize,java hibernate,JSON等）。一些客户端实现的哈希算法也不一

样。但是，memcached 服务器端的行为总是一致的。
最后，从实现的角度看，memcached 是一个非阻塞的、基于事件的服务器程序。这种架构可以很好地解决 C10K problem，并具有极佳的可扩展性。
可以参考 A Story of Caching ， 这篇文章简单解释了客户端与 memcached 是如何交互的。

2、memcached 最大的优势是什么?

请仔细阅读上面的问题(即 memcached 是如何工作的)。Memcached 最大的好处就是它带来了极佳的水平可扩展性，特别是在一个巨大的系统中。由于客户端自己做了一次哈希，那么我们很容易增加大量 memcached 到集群中。memcached 之间没有相互通信，因此不会增加 memcached 的负载;没有多播协议，不会网络通信量爆炸(implode)。memcached 的集群很好用。内存不够了?增加几台 memcached 吧;CPU 不够用了?再增加几台吧；有多余的内存?在增加几台吧，不要浪费了。
基于 memcached 的基本原则，可以相当轻松地构建出不同类型的缓存架构。除了这篇 FAQ，在其他地方很容易找到详细资料的。

3、memcached 和 MySQL 的query cache 相比有什么优缺点?

把 memcached 引入应用中，还是需要不少工作量的。MySQL 有个使用方便的 query cache，可以自动地缓存 SQL 查询的结果，被缓存的 SQL 查询可以被反复地快速执行。Memcached 与之相比，怎么样呢? MySQL 的 query cache 是集中式的，连接到该 query cache 的 MySQL 服务器都会受益。

- 当您修改表时，MySQL 的query cache 会立刻被刷新(flush)。存储一个 memcached item 只需要很少的时间，但是当写操作很频繁时，MySQL 的 query cache 会经常让所有缓存数据都失效。
- 在多核 CPU 上 MySQL 的 query cache 会遇到扩展问题(scalability issues)。在多核 CPU 上 query cache 会增加一个全局锁(global lock)、由于需要刷新更多的缓存数据，速度会变得更慢。
- 在 MySQL 的 query cache 中，我们是不能存储任意的数据的(只能是 SQL 查询结果)。而利用 memcached，我们可以搭建出各种高效的缓存。比如，可以执行多个独立的查询，构建出一个用户对象(user object)，然后将用户对象缓存到 memcached 中，而 query cache 是 SQL 语句级别的，不可能做到这一点。在小的网站中，query cache 会有所帮助，但随着网站规模的增加，query cache 的弊将大于利。

● query cache 能够利用的内存容量受到 MySQL 服务器空闲内存空间的限制。给数据库服务器增加更多的内存来缓存数据，固然是很好的。但是，有了 memcached，只要您有空闲的内存，都可以用来增加 memcached 集群的规模，然后您就可以缓存更多的数据。

4、memcached 和服务器的 local cache (比如 PHP 的 APC、mmap 文件等）相比，有什么优缺点?

首先，local cache 有许多与上面(query cache)相同的问题。local cache 能够利用的内存容量受到(单台)服务器空闲内存空间的限制。不过，local cache 有一点比 memcached 和 query cache 都要好，那就是它不但可以存储任意的数据，而且没有网络存取的延迟。

● local cache 的数据查询更快。考虑把 highly common 的数据放在 local cache 中吧。如果每个页面都需要加载一些数量较少的数据，考虑把它们放在 local cache 吧。

● local cache 缺少集体失效(group invalidation)的特性，在 memcached 集群中，删除或更新一个key 会让所有的观察者觉察到。但是在 local cache 中我们只能通知所有的服务器刷新 cache(很慢，不具扩展性)，或者仅仅依赖缓存超时失效机制。

● local cache 面临着严重的内存限制，这一点上面已经提到。

5、memcached 的 cache 机制是怎样的?

Memcached 主要的 cache 机制是 LRU 最近最少用)算法+超时失效。当您存数据到 memcached 中，可以指定该数据在缓存中可以呆多久 Which is forever,or some time in the future。如果 memcached 的内存不够用了，过期的 slabs 会优先被替换，接着就轮到最老的未被使用的 slabs

6、memcached 如何实现冗余机制?

不实现!我们对这个问题感到很惊讶。Memcached 应该是应用的缓存层。它的设计本身就不带有任何冗余机制。如果一个 Memcached 节点失去了所有数据，您应该可以从数据源(比如数据库)再次获取到数据。您应该特别注意，您的应用应该可以容忍节点的失效。不要写一些糟糕的查询代码，寄希望于 memcached 来保证一切!如果您担心节点失效会大大加重数据库的负担，那么您可以采取一些办法。比如您可以增加更多的节点(来减少丢失一个节点的影响)，热备节点(在其他节点 down 了的时候接管 IP)等等。

7、memcached 如何处理容错的?

不处理!在 memcached 节点失效的情况下，集群没有必要做任何容错处理。如果发生了节点失效，应对的措施完全取决于用户。节点失效时，下面列出几种方案供您选择：

- 忽略它！失效节点被恢复或替换之前，还有很多其他节点可以应对节点失效带来的影响。
- 把失效的节点从节点列表中移除。做这个操作千万要小心！在默认情况下（余数式哈希算法），客户端添加或移除节点，会导致所有的缓存数据不可用！因为哈希参照的节点列表变化了，大部分 key 会因为哈希值的改变而被映射到(与原来)不同的节点上。
- 启动热备节点，接管失效节点所占用的 IP，这样可以防止哈希紊乱(hashing chaos)。
- 如果希望添加和移除节点，而不影响原先的哈希结果，可以使用一致性哈希算法（consistent hashing)。您可以百度一下一致性哈希算法。支持一致性哈希的客户端已经很成熟，而且被广泛使用。去尝试一下吧!
- 两次哈希(reshing)。当客户端存取数据时，如果发现一个节点（down）了，就再做一次哈希(哈希算法与前一次不同)，重新选择另一个节点(需要注意的时，客户端并没有把 down 的节点从节点列表中移除，下次还是有可能先哈希到它)。如果某个节点时好时坏，两次哈希的方法就有风险了，好的节点和坏的节点上都可能存在脏数据(stale data)。

8、如何将 memcached 中 item 批量导入导出?

您不应该这样做!Memcached 是一个非阻塞的服务器。任何可能导致 memcached 暂停或瞬时拒绝服务的操作都应该值得深思熟虑。向 memcached 中批量导入数据往往不是您真正想要的!想象看，如果缓存数据在导出导入之间发生了变化，您就需要处理脏数据了;如果缓存数据在导出导入之间过期了，您又怎么处理这些数据呢?

因此，批量导出导入数据并不像您想象中的那么有用。不过在一个场景倒是很有用。如果您有大量的从不变化的数据，并且希望缓存很快热(warm)起来，批量导入缓存数据是很有帮助的。虽然这个场景并不典型，但却经常发生，因此我们会考虑在将来实现批量导出导入的功能。

Steven Grimm，一如既往地，在邮件列表中给出了另一个很好的例子: `http://lists.danga.com/pipermail/memcached/2007-July/004802.html` 。

9、我需要把 memcached 中的 item 批量导出导入，怎么办?

好吧好吧。如果您需要批量导出导入，最可能的原因一般是重新生成缓存

数据需要消耗很长的时间，或者数据库坏了让您饱受痛苦。
如果一个 memcached 节点 down 了让您很痛苦，那么您还会陷入其他很多麻烦。您的系统太脆弱了。您需要做一些优化工作。比如处理“惊群”问题(比如 memcached 节点都失效了，反复的查询让您的数据库不堪重负...这个问题在 FAQ 的其他提到过)，或者优化不好的查询。记住，Memcached 并不是您逃避优化查询的借口。
如果您的麻烦仅仅是重新生成缓存数据需要消耗很长时间(15 秒到超过 5 分钟)，您可以考虑重新使用数据库。这里给出一些提示:
● 使用 MogileFS (或者 CouchDB 等类似的软件)在存储 item。把 item 计算出来并 dump 磁盘上。MogileFS 可以很方便地覆写 item，并提供快速地访问。您甚至可以把 MogileFS 中的 item 缓存在 memcached 中，这样可以加快读取速度。MogileFS+Memcached 的组合可以加快缓存不命中时的响应速度，提高网站的可用性。
● 重新使用 MySQL。MySQL 的 InnoDB 主键查询的速度非常快。如果大部分缓存数据都可以放到 VARCHAR 字段中，那么主键查询的性能将更好。从 memcached 中按 key 查询几乎等价于 MySQL 的主键查询:将 key 哈希到 64-bit 的整数，然后将数据存储到 MySQL 中。您可以把原始(不做哈希)的 key 存储都普通的字段中，然后建立二级索引来加快查询...key 被动地失效，批量删除失效的 key，等等。
● 上面的方法都可以引入 memcached，在重启 memcached 的时候仍然提供很好的性能。由于您不需要当心”hot”的 item 被 memcached LRU 算法突然淘汰，用户再也不用花几分钟来等待重新生成缓存数据（当缓存数据突然从内存中消失时），因此上面的方法可以全面提高性能。

10、memcached 是如何做身份验证的?

没有身份认证机制!memcached 是运行在应用下层的软件(身份验证应该是应用上层的职责)。memcached 的客户端和服务器端之所以是轻量级的，部分原因就是完全没有实现身份验证机制。这样，memcached 可以很快地创建新连接，服务器端也无需任何配置。
如果您希望限制访问，您可以使用防火墙，或者让 memcached 监听 unix domain socket。

11、memcached 的多线程是什么?如何使用它们?

线程就是定律(threads rule)!在 Steven Grimm 和 Facebook 的努力下，memcached 1.2 及更高版本拥有了多线程模式。多线程模式允许 memcached 能够充分利用多个 CPU，并在 CPU 之间共享所有的缓存数

据。memcached 使用一种简单的锁机制来保证数据更新操作的互斥。相比在同一个物理机器上运行多个 memcached 实例，这种方式能够更有效地处理 multi gets。
如果您的系统负载并不重，也许您不需要启用多线程工作模式。如果您在运行一个拥有大规模硬件的、庞大的网站，您将会看到多线程的好处。
简单地总结一下:命令解析(memcached 在这里花了大部分时间)可以运行在多线程模式下。memcached 内部对数据的操作是基于很多全局锁的(因此这部分工作不是多线程的)。未来对多线程模式的改进，将移除大量的全局锁，提高 memcached 在负载极高的场景下的性能。

12、memcached 能接受的 key 的最大长度是多少?

key 的最大长度是 250 个字符。需要注意的是，250 是 memcached 服务器端内部的限制，如果您使用的客户端支持”key 的前缀”或类似特性，那么key(前缀+原始 key）的最大长度是可以超过250个字符的。我们推荐使用使用较短的 key ，因为可以节省内存和带宽。

13、memcached 对 item 的过期时间有什么限制?

过期时间最大可以达到 30 天.memcached 把传入的过期时间(时间段)解释成时间点后，一旦到了这个时间点，memcached 就把 item 置为失效状态。这是一个简单但 obscure 的机制。

14、memcached 最大能存储多大的单个 item?

1MB。如果你的数据大于 1MB，可以考虑在客户端压缩或拆分到多个 key 中。