# 1.Spring

## 1.什么是spring?

Spring 是个java[企业级](https://www.baidu.com/s?wd=%E4%BC%81%E4%B8%9A%E7%BA%A7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/ying847782627/article/details/_blank)应用的开源开发框架。Spring主要用来开发Java应用，但是有些扩展是针对构建J2EE平台的web应用。Spring 框架目标是简化Java企业

## 2.使用Spring框架的好处是什么？

轻量：Spring 是轻量的，基本的版本大约2MB。

控制反转：Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。

面向切面的编程(AOP)：Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。

容器：Spring 包含并管理应用中对象的生命周期和配置。

MVC框架：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。

事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。

异常处理：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。级应用开发，并通过POJO为基础的编程模型促进良好的编程习惯。

## 3.Spring由哪些模块组成?

以下是Spring 框架的基本模块：Core module、Bean module、Context module、Expression Language module、JDBC module、ORM module、OXM module、Java Messaging Service(JMS) module、Transaction module、Web module、Web-Servlet module、Web-Struts module、Web-Portlet module

## 4.什么是Spring IOC 容器以及优点？

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和JNDI查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC容器支持加载服务时的饿汉式初始化和懒加载。

答：即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，**Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。**

**IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。**

## 5.什么是Spring的依赖注入？

依赖注入，是IOC的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC容器）负责把他们组装起来。

## 6.有哪些不同类型的IOC（依赖注入）方式？

****构造器依赖注入：****构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

## 7.什么是Spring beans?

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中<bean/> 的形式定义。

Spring 框架定义的beans都是单件beans。在bean tag中有个属性”singleton”，如果它被赋为TRUE，bean 就是单件，否则就是一个 prototype bean。默认是TRUE，所以所有在Spring框架中的beans 缺省都是单件。

****Setter方法注入：****Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

## 8.你怎样定义类的作用域?

当定义一个<bean> 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

## 9.解释Spring支持的几种bean的作用域。

Spring框架支持以下五种bean的作用域：

****singleton :****bean在每个Spring ioc 容器中只有一个实例。

****prototype****：一个bean的定义可以有多个实例。

****request****：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。

****session****：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

****global-session****：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

缺省的Spring bean 的作用域是Singleton.

## 10.解释Spring框架中bean的生命周期。

Spring容器 从XML 文件中读取bean的定义，并实例化bean。

Spring根据bean的定义填充所有的属性。

如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。

如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。

如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。

如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。

如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。

如果bean实现了 DisposableBean，它将调用destroy()方法。

## 11.什么是Spring的内部bean？

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

## 12.在 Spring中如何注入一个java集合？

Spring提供以下几种集合的配置元素：

<list>类型用于注入一列值，允许有相同的值。<set> 类型用于注入一组值，不允许有相同的值。<map> 类型用于注入一组键值对，键和值都可以为任意类型。<props>类型用于注入一组键值对，键和值都只能为String类型。

## 13.什么是bean的自动装配？

Spring 容器能够自动装配相互合作的bean，这意味着容器不需要<constructor-arg>和<property>配置，能通过Bean工厂自动处理bean之间的协作。

## 14.自动装配有哪些局限性 ?

****重写****： 你仍需用 <constructor-arg>和 <property> 配置来定义依赖，意味着总要重写自动装配。

****基本数据类型****：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。

****模糊特性：****自动装配不如显式装配精确，如果有可能，建议使用显式装配。

## 15.什么是基于Java的Spring注解配置? 给一些注解的例子.

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

## 16.@Required 注解

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。

## 17. @Autowired 注解

@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

## 18. @Qualifier 注解

当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的bean。

## 19.Spring支持的事务管理类型：

spring支持编程式事务管理和声明式事务管理两种方式：

（1）编程式事务管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

（2）声明式事务管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

（3）显然声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式。声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

## 20.你更倾向用那种事务管理类型？

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

## 21.Spring事务的隔离级别

隔离级别定义一个事务可能接受其他并发事务活动受影响的程度。也可以想象成事务对于数据处理的自私程度。它是多个事务并发执行时，产生的。

多个事务同时运行，经常会为了完成他们的工作而操作同一个数据。并发虽然是必需的，但是会导致以下问题：

* 胀读——A事务读取数据并修改，未提交之间B事务又读取了数据
* 不可重复读——A事务读取数据，B事务读取数据并修改，A事务再读取数据时发现两次数据不一致
* 幻读——事务A读取或删除了全部数据，事务B又insert一条数据，这时事务A发现莫名其妙的多了一条数据

理想状态下，事务之间是完全隔离的。但是完全隔离会影响性能，因为隔离的实现依赖于数据库中的锁，侵占性锁会阻碍并发，要求事务互相等待。

考虑到完全隔离会影响性能，而且并不是所有的情况都要求完全隔离，所以有时候可以在事务隔离方面灵活处理。因此，就有好几个隔离级别。

## 22.Spring事务管理的隔离级别

解决事务的安全性问题：脏读、不可重复读、幻读。下面是Spring事务的隔离级别：

| **隔离级别** | **含义** |
| --- | --- |
| DEFAULT | 使用数据库默认的隔离级别（spring中的选择项） |
| READ\_UNCOMMITTED | 允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读 |
| READ\_COMMITTED | 允许在并发事务已经提交后读取。可防止脏读， 但幻读、不可重复读仍可能发生（Oracle默认的隔离级别） |
| REPEATABLE\_READ | 对相同字段的多次读取是一致的，除数据对事务本身改变。可防止脏、不可重复读，但幻读仍可能发生 （MySQL默认的隔离级别） |
| SERIALIZABLE | 完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。 |

## 23.Spring事务的传播行为

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。在TransactionDefinition定义中包括了如下7个表示传播行为的常量：

| **传播行为** | **说明** |
| --- | --- |
| PROPAGATION\_REQUIRED | 业务方法需要在一个事务中运行。如果方法运行时，已经处在一个事务中，那么加入到该事务，否则为自己创建一个新的事务 |
| PROPAGATION\_NOT\_SUPPORTED | 声明方法不需要事务。如果方法没有关联到一个事务，容器不会为它开启事务。如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务便会恢复执行 |
| PROPAGATION\_REQUIRES\_NEW | 属性表明不管是否存在事务，业务方法总会为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务会被挂起，新的事务会被创建，直到方法执行结束，新事务才算结束，原先的事务才会恢复执行 |
| PROPAGATION\_MANDATORY | 该属性指定业务方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果业务方法在没有事务的环境下调用，容器就会抛出异常。 |
| PROPAGATION\_SUPPORTS | 这一事务属性表明，方法可以受事务控制，也可以不。如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行 |
| PROPAGATION\_NEVER | 指定业务方法绝对不能在事务范围内执行。如果业务方法在某个事务中执行，容器会抛出异常，只有业务方法没有关联到任何事务，才能正常执行 |
| PROPAGATION\_NESTED | 如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按REQUIRED属性执行.它使用了一个单独的事务， 这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效 |

注意REQUIRES\_NEW和NESTED两者的区别；  
PROPAGATION\_REQUIRES\_NEW启动一个新的, 不依赖于环境的 "内部" 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行。  
  
PROPAGATION\_NESTED 开始一个 "嵌套的" 事务, 它是已经存在事务的一个真正的子事务. 潜套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交。

spring事务的传播行为说的是当一个方法调用另一个方法时，事务该如何操作。  
（1）PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

（2）PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

（3）PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

（4）PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

（5）PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

（6）PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

（7）PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

24.对嵌套事务的理解？  
嵌套：是子事务在父事务中执行，子事务是父事务的一部分，在进入子事务之前，父事务建立一个回滚点，叫save point，然后执行子事务，这个子事务也算是父事务的一部分，然后子事务执行结束，父事务继续执行。下面看几个问题就懂了

如果子事务回滚，会发生什么？  
答：父事务会回滚到save point，然后尝试其他的事务或者其他的业务逻辑，父事务之前的操作不会受到影响，更不会自动回滚

如果父事务回滚，会发生什么？  
答：父事务回滚，子事务也会跟着回滚。父事务结束之前，子事务不会提交。

嵌套事务的提交是什么情况？  
答：子事务是父事务的一部分，所以子事务先提交，父事务再提交

## 25.事务超时

假设事务的运行时间变得格外的长，由于事务可能涉及对后端数据库的锁定，所以长时间运行的事务会不必要地占用数据库资源，这时就可以声明一个事务在特定时间后自动回滚。

由于超时时钟在一个事务启动的时候开始的，因此，有对于那些具有可能启动一个新事物的传播行为（PROPAGATION\_REQUIRED，PROPAGATION\_REQUIRES\_NEW，PROPAGATION\_NESTED）的方法来说，声明事务超时才有意义，默认为30S。

## 26.是否只读

如果事务只对后端数据进行读操作，那么如果将事务设置为只读事务，可以利用后端数据库优化措施进行适当优化。

只读事务”并不是一个强制选项，它只是一个“暗示”，提示数据库驱动程序和数据库系统，这个事务并不包含更改数据的操作，那么JDBC驱动程序和数据库就有可能根据这种情况对该事务进行一些特定的优化，比方说不安排相应的数据库锁，以减轻事务对数据库的压力，毕竟事务也是要消耗数据库的资源的。但是你非要在“只读事务”里面修改数据，也并非不可以，只不过对于数据一致性的保护不像“读写事务”那样保险而已。

因此，“只读事务”仅仅是一个性能优化的推荐配置而已，并非强制你要这样做不可。

只读事务实在开启事务时有数据库实施的，所以只对具备启动新事务的传播机制有效，如REQUIRED、REQUIRES\_NEW、NESTED。

## 27.回滚规则

回滚规则定义了哪些异常引起回滚，哪些不引起。在默认情况下，事务只出现运行时异常（Runtime Exception）时回滚，而在出现受检查异常（Checked Exception）时不回滚。

但是我们可以在Spring中进行定义来改变其默认行为。Spring在xml文件配置事务时提供了rollback-for和no-rollback-for参数，来指定回滚和不会滚的异常名称，该名称对应的类为Throwable的子类。

## 28.解释AOP

答：AOP即面向切面编程，是OOP编程的有效补充。使用AOP技术，可以将一些系统性相关的编程工作，独立提取出来，独立实现，然后通过切面切入进系统。从而避免了在业务逻辑的代码中混入很多的系统相关的逻辑——比如权限管理，事物管理，日志记录等等。

AOP分为静态AOP和动态AOP：

* 静态AOP是指AspectJ实现的AOP，他是将切面代码直接编译到Java类文件中。
* 动态AOP是指将切面代码进行动态织入实现的AOP，JDK动态代理。

## 29.Aspect 切面

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

## 30. 在Spring AOP 中，关注点和横切关注的区别是什么？

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

## 31. 连接点

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

## 32.通知有哪些类型？

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。  
（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。  
（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。  
（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。  
环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

## 谈谈你对Spring的AOP理解：

AOP，一般称为面向方面（切面）编程，作为面向对象的一种补充，用于解剖封装好的对象内部，找出其中对多个对象产生影响的公共行为，并将其封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），切面将那些与业务无关，却被业务模块共同调用的逻辑提取并封装起来，减少了系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。  
AOP实现的关键在于AOP框架自动创建的AOP代理，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：  
①JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法，当我们调用代理类对象的方法时，这个“调用”会转送到invoke方法中，代理类对象作为proxy参数传入，参数method标识了我们具体调用的是代理类的哪个方法，args为这个方法的参数。  
②如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法，覆盖方法时可以添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

## 34.Spring如何处理线程并发问题？

Spring使用ThreadLocal解决线程安全问题。

我们知道在一般情况下，只有有状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean(如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等)中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

1. 在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放[对象锁](https://www.baidu.com/s?wd=%E5%AF%B9%E8%B1%A1%E9%94%81&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/weixin_37327955/article/details/_blank)等繁杂的问题，程序设计和编写难度相对较大。
2. （2）而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

（3）概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

## 35.Spring 框架中都用到了哪些设计模式？

（1）代理模式—在AOP和remoting中被用的比较多。

（2）单例模式—在spring配置文件中定义的bean默认为单例模式。

（3）工厂模式—BeanFactory用来创建对象的实例。

（4）模板方法—用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

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

（6）视图帮助(View Helper )—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。

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

## 36.解释一下Spring AOP里面的几个名词：

（1）切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是J2EE应用中一个关于横切关注点的很好的例子。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @Aspect 注解（@AspectJ风格）来实现。

（2）连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在Spring AOP中，一个连接点 总是 代表一个方法的执行。 通过声明一个org.aspectj.lang.JoinPoint类型的参数可以使通知（Advice）的主体部分获得连接点信息。

（3）通知（Advice）：在切面的某个特定的连接点（Joinpoint）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。

（4）切入点（Pointcut）：匹配连接点（Joinpoint）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。 切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。

（5）引入（Introduction）：（也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。 Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（advised） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：把切面（aspect）连接到其它的应用程序类型或者对象上，并创建一个被通知（advised）的对象。 这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。 Spring和其他纯Java AOP框架一样，在运行时完成织入。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

## 37.Spring中bean的加载过程：

（1）获取配置文件资源；

（2）对获取的xml资源进行一定的处理检验；

（3）处理包装资源；

（4）解析处理包装过后的资源；

（5）加载提取bean并注册(添加到beanDefinitionMap中)。

## 38.Spring 控制器的加载过程：（XML版）

（1）Web容器创建；

（2）上下文创建，但未初始化；

（3）[监听器](https://www.baidu.com/s?wd=%E7%9B%91%E5%90%AC%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/weixin_37327955/article/details/_blank)创建，并注册到Context上；

（4）上下文初始化；

（5）通知到监听者，Spring配置文件/@Configuration加载；

（6）Load-on-startup>0的ServletConfig创建，springMVC的[DispatcherServlet](https://www.baidu.com/s?wd=DispatcherServlet&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/weixin_37327955/article/details/_blank)此时创建。

PS：Spring容器时SpringMVC的父容器。Spring的AOP在Spring的上下文创建时就会创建；如果想要代理SpringMVC的控制层，需要将配置写到SpringMVC的配置文件下。

# 2.Spring MVC

## 1. 什么是Spring的MVC框架？

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

## 2. **[DispatcherServlet](https://www.baidu.com/s?wd=DispatcherServlet&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/ying847782627/article/details/_blank)**

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

## 3. WebApplicationContext

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

## 4. 什么是Spring MVC框架的控制器？

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

## 5. @Controller 注解

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

## 6. @RequestMapping 注解

该注解是用来映射一个URL到一个类或一个特定的方处理法上。

## 7.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

## 8.SpringMVC的优点

1.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  
【说明：在springmvc的各个组件中，处理器映射器、处理器适配器、视图解析器称为springmvc的三大组件。需要用户开发的组件有controller、view】  
2.与Spring框架天生整合,无框架兼容问题 与Struts2相比安全性高 配置量小、开发效率高

## 9.json数据交互

**@RequestBody**作用：  
 @RequestBody注解用于读取http请求的内容(字符串)，通过springmvc提供 的HttpMessageConverter接口将读到的内容（json数据）转换为java对象并 绑定到Controller方法的参数上。  
**@ResponseBody**作用：  
 @ResponseBody注解用于将Controller的方法返回的对象，通过springmvc 提供的HttpMessageConverter接口转换为指定格式的数据如：json,xml等， 通过Response响应给客户端

## 10.SpringMVC的工作流程

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，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

# 3.RabbitMQ

## 1.简述RabbitMQ中的几种Exchange的作用。

答：有四种exchange

1）direct exchange：只有当消息的routing key和exchange binding的routing key完全一致的时候，exchange才会将消息发送到binding对应的队列去。

2）topic exchange：当消息的routing key和exchange binding的routing key模糊匹配的时候，exchange才会将消息发送到binding对应的队列去。

3）fanout exchange：和routing key无关，exchange将消息转发给所有binding的队列去。

4）header exchange：和routing key无关，但是需要配置exchange header的匹配参数以及匹配方式（例如any/all等），当匹配了exchange header的参数和方式的时候，exchange将消息转发到binding的队列去。

备注：header中也是key value 的形式的 匹配的时候 根据key value 有相同的发送

## 2.在RabbitMQ中，一个队列上所有的消费者可以消费同一条消息么？

不可以消费同一条消息，如果要消费同一条消息需要使用exchange配置多个Queue进行绑定，配合使用消息的routing key可以实现群发

## 3.如果生产环境有问题，需要查询哪里？

答：查看日志/var/log/rabbitmq/XXX 找到对应的报错信息在进行对应的处理

查docker中Rabbitmq的日志在哪？

## 4.RabbitMQ的集群内同步什么信息？

答：RabbitMQ的集群分为两种，一种是普通模式，一种是镜像模式。默认策略是同步exchange的信息，不同步队列消息，需要增加额外的policy才能同步队列消息。

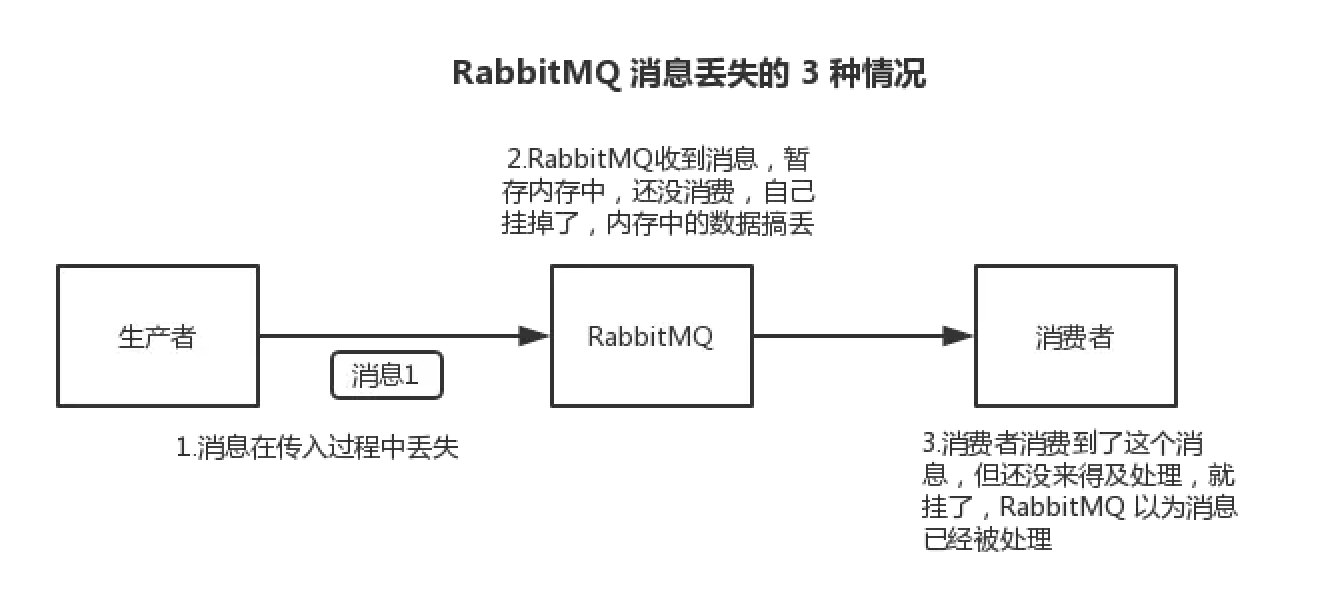
## 5.RabbitMQ的queue和ActiveMQ的queue有什么区别？

答：ActiveMQ的queue只允许一个生产者和一个消费者，RabbitMQ的queue允许多个生产者和多个消费者。

## 6. RabbitMQ的queue和ActiveMQ的topic有什么区别？

答：ActiveMQ的topic会将同一个消息发送给所有订阅的消费者，RabbitMQ的queue只将同一个消息发送给所有订阅的消费者之一。

## Rabbit如何保证消息的可靠性传输（如何处理消息丢失的问题）？



### 7.1 生产者弄丢了数据

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者**发送数据之前**开启 RabbitMQ 事务channel.txSelect，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务channel.txRollback，然后重试发送消息；如果收到了消息，那么可以提交事务channel.txCommit。

|  |
| --- |
| // 开启事务  channel.txSelect  try {  // 这里发送消息  } catch (Exception e) {  channel.txRollback  // 这里再次重发这条消息  }  // 提交事务  channel.txCommit |

但是问题是，RabbitMQ 事务机制（同步）一搞，基本上**吞吐量会下来，因为太耗性能**。

所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 cnofirm 机制最大的不同在于，**事务机制是同步的**，你提交一个事务之后会**阻塞**在那儿，但是 confirm 机制是**异步**的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你的一个接口通知你这个消息接收到了。

所以一般在生产者这块**避免数据丢失**，都是用 confirm 机制的。

### 7.2 RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须**开启 RabbitMQ 的持久化**，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，**恢复之后会自动读取之前存储的数据**，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，**可能导致少量数据丢失**，但是这个概率较小。

设置持久化有**两个步骤**：

* 创建 queue 的时候将其设置为持久化  
  这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是它是不会持久化 queue 里的数据的。
* 第二个是发送消息的时候将消息的 deliveryMode 设置为 2  
  就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

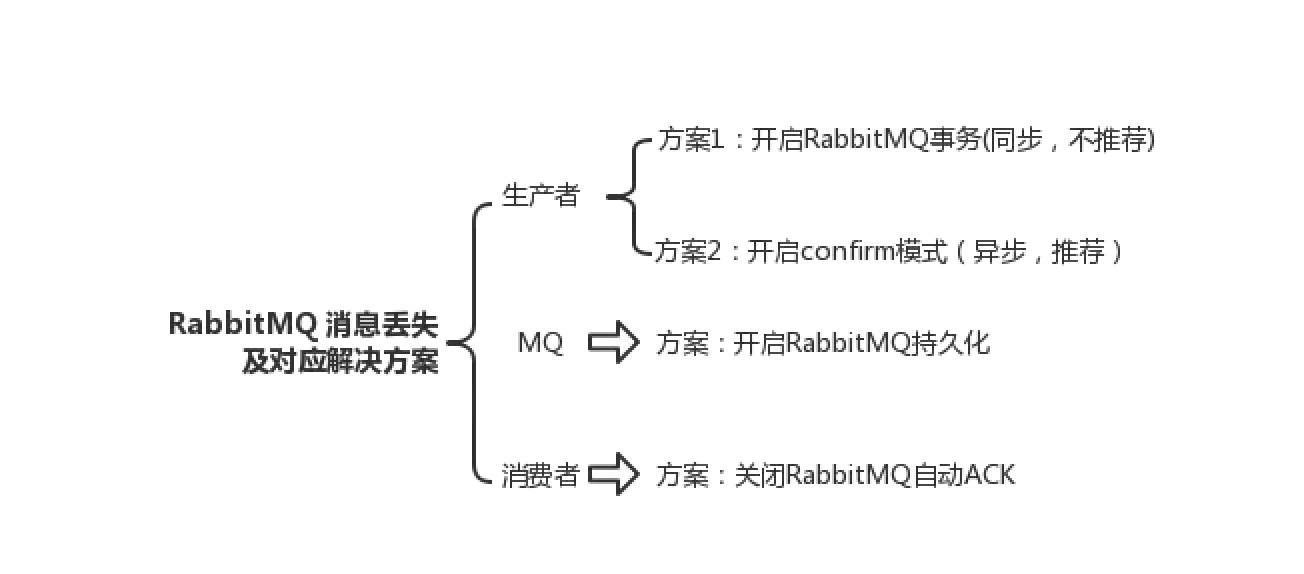
注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失。

所以，持久化可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。

### 7.3消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，**刚消费到，还没处理，结果进程挂了**，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 ack 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 ack 一把。这样的话，如果你还没处理完，不就没有 ack 了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



## 8.RabbitMQ消息基于什么传输？

由于TCP连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ使用信道的方式来传输数据。信道是建立在真实的TCP连接内的虚拟连接，且每条TCP连接上的信道数量没有限制。

## 9.消息怎么路由？

消息提供方->路由->一至多个队列

消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。

消息到达交换器后，RabbitMQ会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）；

交换器分四中其中常用的交换器主要分为以下三种：

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用topic交换器时，可以使用通配符

## 10.如何确保消息接收方消费了消息？

接收方消息确认机制：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ仅通过Consumer的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ给了Consumer足够长的时间来处理消息。

下面罗列几种特殊情况：

* 如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要根据bizId去重）
* 如果消费者接收到消息却没有确认消息，连接也未断开，则RabbitMQ认为该消费者繁忙，将不会给该消费者分发更多的消息。

## 11.如何确保消息正确地发送至RabbitMQ？

RabbitMQ使用发送方确认模式，确保消息正确地发送到RabbitMQ。

发送方确认模式：将信道设置成confirm模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的ID。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一ID）。如果RabbitMQ发生内部错误从而导致消息丢失，会发送一条nack（not acknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

## 12.为什么要引入MQ系统，直接读写数据库不行吗？

其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么？

面试官问你这个问题，期望的一个回答是说，你们公司有个什么业务场景，这个业务场景有个什么技术挑战，如果不用 MQ 可能会很麻烦，但是你现在用了 MQ 之后带给了你很多的好处。

先说一下消息队列常见的使用场景吧，其实场景有很多，但是比较核心的有 3 个：解耦、异步、削峰。

解耦：多系统多进程的数据交换，用pub/sub

异步：把大数据量的同步处理改为异步

削峰：一般的A 系统使用 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。如果使用 MQ， 每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最 大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉，这又涉及请求排队的问题。

## 13.如何解决消息队列的延时以及过期失效问题？

其实本质针对的场景，都是说，可能你的消费端出了问题，不消费了；或者消费的速度极其慢，造成消息堆积了，MQ存储快要爆了，甚至开始过期失效删除数据了。

针对这个问题可以有事前、事中、事后三种处理

事前：开发预警程序，监控最大的可堆积消息数，超过就发预警消息（比如短信），不要等出生产事故了再处理。

事中：看看消费端是不是故障停止了，紧急重启。

事后：需要对消费端紧急扩容 ，增加处理消费者进程，如扩充10倍处理，但其实这也有个问题，即数据库的吞吐是有限制的，如果是消费到数据库也是没办法巨量扩容的，所以还是要在吞吐能力支持下老老实实的泄洪消费。

所以事前预防还是最重要的。否则出发删除过期数据。

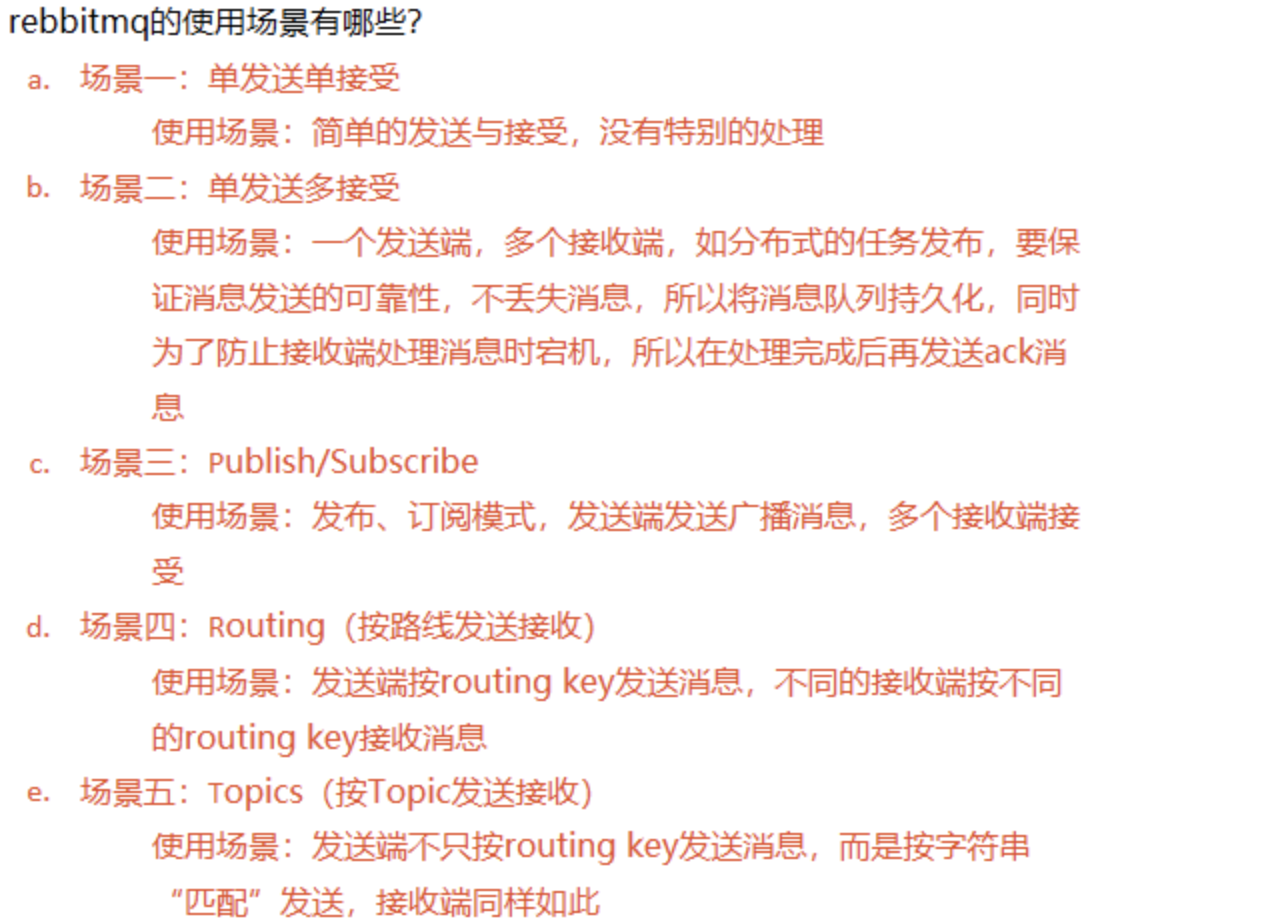
## RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？ 答：可以认为是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。

## 为什么不应该对所有的 message 都使用持久化机制？

答：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。

## 16.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。

1. 

## 17.消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。  
通过路由可实现多消费的功能

## 18.如何避免消息重复投递或重复消费？

在消息生产时，MQ内部针对每条生产者发送的消息生成一个inner-msg-id，作为去重的依据（消息投递失败并重传），避免重复的消息进入队列；

在消息消费时，要求消息体中必须要有一个bizId（对于同一业务全局唯一，如支付ID、订单ID、帖子ID等）作为去重的依据，避免同一条消息被重复消费。

## 19.ActiveMQ与RabbitMQ的区别

1. ActiveMQ/ApolloMQ

　　优点：老牌的消息队列，使用Java语言编写。对JMS支持最好，采用多线程并发，资源消耗比较大。如果你的主语言是Java，可以重点考虑。

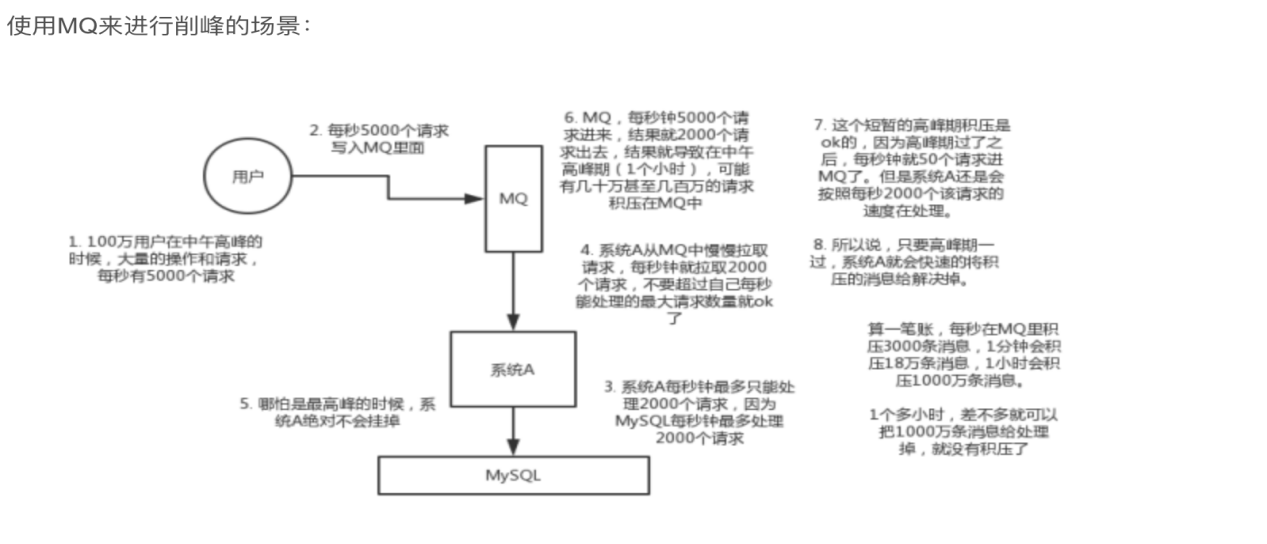
　　缺点：由于历史悠久，历史包袱较多，版本更新很缓慢。集群模式需要依赖Zookeeper实现。最新架构的产品被命名为Apollo，号称下一代ActiveMQ，目前案例较少。

2. RabbitMQ

　　优点：生态丰富，使用者众，有很多人在前面踩坑。AMQP协议的领导实现，支持多种场景。淘宝的MySQL集群内部有使用它进行通讯，OpenStack开源云平台的通信组件，最先在金融行业得到运用。

缺点：Erlang代码你Hold得住不? 虽然Erlang是天然集群化的，但RabbitMQ在高可用方面做起来还不是特别得心应手，别相信广告。

## 20.MQ销峰场景



# 4.SpringBoot

## 1.Spring Boot、Spring MVC 和 Spring 有什么区别？

**1.1、Spring**

Spring最重要的特征是依赖注入。所有 SpringModules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

**1.2、Spring MVC**

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。

**1.3、SpringBoot**

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

Spring Boot 通过一个自动配置和启动的项来目解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

## 2.什么是 Spring Boot Stater ？

启动器是一套方便的依赖没描述符，它可以放在自己的程序中。你可以一站式的获取你所需要的 Spring 和相关技术，而不需要依赖描述符的通过示例代码搜索和复制黏贴的负载。

比如我么你项目中使用 Sping data JPA 访问数据库，我们只需要在项目中的pom.xml文件中引入 spring-boot-starter-data-jpa 依赖项，你就可以完美进行。

## 你能否举一个例子来解释更多 Staters 的内容？

让我们来思考一个 Stater 的例子 -Spring Boot Stater Web。

如果你想开发一个 web 应用程序或者是公开 REST 服务的应用程序。Spring Boot Start Web 是首选。让我们使用 Spring Initializr 创建一个 Spring Boot Start Web 的快速项目。

依赖项可以被分为：

* Spring - core，beans，context，aop
* Web MVC - （Spring MVC）
* Jackson - for JSON Binding
* Validation - Hibernate,Validation API
* Enbedded Servlet Container - Tomcat
* Logging - logback,slf4j

任何经典的 Web 应用程序都会使用所有这些依赖项。Spring Boot Starter Web 预先打包了这些依赖项。

作为一个开发者，我不需要再担心这些依赖项和它们的兼容版本。

## Spring Boot 还提供了其它的哪些 Starter Project Options？

* Spring Boot 也提供了其它的启动器项目包括，包括用于开发特定类型应用程序的典型依赖项。
* spring-boot-starter-web-services - SOAP Web Services；
* spring-boot-starter-web - Web 和 RESTful 应用程序；
* spring-boot-starter-test - 单元测试和集成测试；
* spring-boot-starter-jdbc - 传统的 JDBC；
* spring-boot-starter-hateoas - 为服务添加 HATEOAS 功能；
* spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权；
* spring-boot-starter-data-jpa - 带有 Hibeernate 的 Spring Data JPA；
* spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务；

## 5.Spring2 和 Spring5 所需要的最低 Java 版本是什么？

Spring Boot 2.0 需要 Java8 或者更新的版本。Java6 和 Java7 已经不再支持。

## 6.什么是嵌入式服务器？我们为什么要使用嵌入式服务器呢?

思考一下在你的虚拟机上部署应用程序需要些什么。

第一步： 安装 Java

第二步： 安装 Web 或者是应用程序的服务器（Tomat/Wbesphere/Weblogic 等等）

第三步： 部署应用程序 war 包

如果我们想简化这些步骤，应该如何做呢？

让我们来思考如何使服务器成为应用程序的一部分？

你只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了，

是不是很爽？

这个想法是嵌入式服务器的起源。

当我们创建一个可以部署的应用程序的时候，我们将会把服务器（例如，tomcat）嵌入到可部署的服务器中。

例如，对于一个 Spring Boot 应用程序来说，你可以生成一个包含 Embedded Tomcat 的应用程序 jar。你就可以想运行正常 Java 应用程序一样来运行 web 应用程序了。

嵌入式服务器就是我们的可执行单元包含服务器的二进制文件（例如，tomcat.jar）。

## 7.我看你上面写了熟悉 Spring Boot，那你能讲下为什么我们要用 Spring Boot 吗？

下面我列几个最常见的三个回答：

**A：Spring Boot 最主要是不用 XML 配置，可以用 Java 来配置 bean，省去了许多配置文件。**

我又问：Spring 本身就可以用 Java 配置代替 XML 配置，和 Spring Boot 有什么关系呢？

**应该回答：其实SpringBoot最重要的概念就是自动配置这些依赖的配置文件，我们只需要写他们用到的资源文件（yml文件），省去了很多重复操作。让管理xml的工作变得简单。还有一点就是springBoot可以帮我们排除Jar包冲突。**

**B：Spring Boot 我们用来做 Spring Cloud 微服务。**

我又问：微服务和 Spring Boot 有什么关系？不用 Spring Boot 行不行？

然后对方就吱吱唔唔了……

**应该回答：微服务是一种架构理念与springBoot没什么关系，由于实现微服务可以使用dubbo和springCloud两种比较常见的解决方案，当然不用这些框架我们也可以使用SSH、SSM框架实现，需要把项目架构形成颗粒感，自己实现调用方式，但是实现起来不太容易需要写很多代码。因为Cloud基于Boot实现的依赖，如果使用springCloud就必须使用springBoot。**

**C：Spring Boot 可以打 jar 包部署，内部集成了Tomcat。**

这个确实是 Spring Boot 的特色，但是我还是觉得没有答到关键点上。

我们可以回答Spring Boot 最重要的功能是：**自动配置**。

Spring Boot 的开启注解是：@SpringBootApplication，其实它就是由下面三个注解组成的：

@Configuration

@ComponentScan

@EnableAutoConfiguration

上面三个注解，前面两个都是 Spring 自带的，和 Spring Boot 无关，所以说上面的回答的不是在点上

Spring Boot 最最核心的就是这个 @EnableAutoConfiguration 注解了，它能根据类路径下的 jar 包和配置动态加载配置和注入bean。

# ==================================

# 1. Java 基础

1. JDK 和 JRE 有什么区别？

简单来说，JRE可以支撑Java程序的运行，包括JVM虚拟机（java.exe等）和基本的类库（rt.jar等），JDK可以支持Java程序的开发，包括编译器（javac.exe）、开发工具（javadoc.exe、jar.exe、keytool.exe、jconsole.exe）和更多的类库（如tools.jar）等。

1. == 和 equals 的区别是什么？

java中的数据类型，可分为两类：   
1.基本数据类型，也称原始数据类型。byte,short,char,int,long,float,double,boolean   
  他们之间的比较，应用双等号（==）,比较的是他们的值。   
2.复合数据类型(类)   
  当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。 JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。  
  对于复合数据类型之间进行equals比较，在没有覆写equals方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为Object的equals方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。

1. 两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗？

两者之间唯一的必然关系被说反了，equls返回为true,则两者的hashcode一定相等，意思即相等的对象必须具有相等的[哈希码](https://www.baidu.com/s?wd=%E5%93%88%E5%B8%8C%E7%A0%81&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)。每当equals方法被覆写，通常需要重写hashCode方法从而保持对象行为的一致性。而具有相等的hashcode的两个对象equals不一定成立。

## final 在 java 中有什么作用？

**final关键字的作用**

相信对于final的用法，大多数人都可以随口说出三句话：

1、被final修饰的类不可以被继承

2、被final修饰的方法不可以被重写

**3、被final修饰的变量不可以被改变**

重点就是第三句。被final修饰的变量不可以被改变，什么不可以被改变呢，是变量的引用？还是变量里面的内容？还是两者都不可以被改变？写个例子看一下就知道了：

1 public class FinalString 2 {  
3 private String str; 4  
5 public FinalString(String str) 6 {  
7 this.str = str; 8 }  
9  
10 public String getStr() 11 { 12 return str; 13 } 14  
15 public void setStr(String str) 16 { 17 this.str = str; 18 } 19 }

1 public class Test 2 { 3 public static void main(String[] args) 4 { 5 final FinalString fs = new FinalString("1"); 6 fs.setStr("2"); 7 System.out.println(fs.getStr()); 8 } 9 }

运行一下，一点问题都没有。稍微修改一下呢：

1 public static void main(String[] args)  
2 {  
3 final FinalString fs = new FinalString("1");  
4 final FinalString fss = new FinalString("333");  
5 fs = fss;  
6 }

第5行报错了，“The final local variable fs cannot be assigned”。可见，**被final修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的**。OK，那final修饰数组呢？

1 public static void main(String[] args)  
2 {  
3 final String[] strs0 = {"123","234"};  
4 final String[] strs1 = {"345","456"};  
5 strs1 = strs0;  
6 strs1[1] = "333";  
7 }

同样，第5行报错了“The final local variable strs1 cannot be assigned”，第6行一点问题都没有。变量和数组一样，都是**引用不可变，引用指向的内容可变**。实际上如果用过FindBugs插件的应该知道，假如代码里面用final修饰了一个数组，那么改行代码会被作为findBugs的一个bug被查找出来，因为“用final修饰数组是没有意义的”。

接下来，再看一下用final修饰方法参数的场景：

1 public class Test 2 {  
3 public static void main(String[] args) 4 {  
5 FinalString fs = new FinalString("");  
6 A(fs);  
7 }  
8  
9 private static void A(final FinalString fs) 10 { 11 fs.setStr("123"); 12 FinalString fss = new FinalString("22"); 13 fs = fss; 14 } 15 }

一样，同样是13行报错，11行没有问题，相信大家已经知道原因了。

**总结**

“引用”是Java中非常重要的一个概念，对于引用的理解不深，很容易犯一些自己都没有意识到的错误。被final修饰的变量，不管变量是在是哪种变量，切记**不可变的是变量的引用而非引用指向对象的内容**。另外，本文中关于final的作用还有两点没有讲到：

1、被final修饰的方法，JVM会尝试为之寻求内联，这对于提升Java的效率是非常重要的。因此，假如能确定方法不会被继承，那么尽量将方法定义为final的

2、被final修饰的常量，在编译阶段会存入调用类的常量池中，具体参见[类加载机制]

## 5. java 中的 Math.round(-11.5) 等于多少？

round是[四舍五入](https://www.baidu.com/s?wd=%E5%9B%9B%E8%88%8D%E4%BA%94%E5%85%A5&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)，注意负数5是舍的，例如：Math.round(1.5)值是2，Math.round(-1.5)值是-1；  
floor就是直接去掉小数保留整数，即如果参数是正数则小数部分全舍，参数是负数则小数部分全入。 例如：Math.floor(2.6)的值是2，Math.floor(-2.1)的值是-3。

个人理解：round()方法可以这样理解：将括号内的数+0.5之后，向下取值，

比如：round(3.4)就是3.4+0.5=3.9，向下取值是3，所以round(3.4)=3;

round(-10.5)就是-10.5+0.5=-10，向下取值就是-10，所以round(-10.5)=-10

所以，Math.round(11.5)=12;

## 6.String 属于基础的数据类型吗？

基本数据类型包括byte、int、char、long、float、double、boolean和short。

对应的基本类型包装类：Boolean Byte Character Short Integer Long Float Double

java.lang.String类是final类型的，因此不可以继承这个类、不能修改这个类。为了提高效率节省空间，我们应该用StringBuffer类 （注意：这地方有可能问stringbuffer和stringbuilder的区别）

## 7. java 中操作字符串都有哪些类？它们之间有什么区别？

## 8. String str="i"与 String str=new String(“i”)一样吗？

不一样。前者是一个常量，后者又重新new了一个对象，内存空间不一样。

## 9.如何将字符串反转？

第一种：

public class Main {

public static void main(String[] args) {

String s1 = "asdfghjkl"; System.out.println(new Main().swapWords(s1));

}

public void swap(char[] arr, int begin, int end) {

while (begin < end) {

char temp = arr[begin];

arr[begin] = arr[end];

arr[end] = temp;

begin++;

end--;

}

}

public String swapWords(String str) {

char[] arr = str.toCharArray();

swap(arr, 0, arr.length - 1);

int begin = 0;

for (int i = 1; i < arr.length; i++) {

if (arr[i] == ' ') {

swap(arr, begin, i - 1);

begin = i + 1;

}

}

return new String(arr);

}

}

第二种：

public class Main {

public static void main(String[] args) {

String s1 = "asdfghjkl";

String[] s = s1.split("");

List<String> list = list = Arrays.asList(s);

Collections.reverse(list);

System.out.println(list);

}

}

第三种：

public class Main {

public static void main(String[] args) {

String s1 = "asdfghjkl";

System.out.println(new StringBuilder(s1).reverse().toString());

}

}

## String 类的常用方法都有那些？

1、和长度有关：

* int length() 得到一个字符串的字符个数

2、和数组有关：

* byte[] getByte() ) 将一个字符串转换成字节数组
* char[] toCharArray() 将一个字符串转换成字符数组
* String split(String) 将一个字符串按照指定内容劈开

3、和判断有关：

* boolean equals() 判断两个字符串的内容是否一样
* boolean equalsIsIgnoreCase(String) 忽略太小写的比较两个字符串的内容是否一样
* boolean contains(String) 判断一个字符串里面是否包含指定的内容
* boolean startsWith(String) 判断一个字符串是否以指定的内容开头
* boolean endsWith(String) 判断一个字符串是否以指定的内容结尾

4、和改变内容有关：

* String toUpperCase() 将一个字符串全部转换成大写
* String toLowerCase() 将一个字符串全部转换成小写
* String replace(String,String) 将某个内容全部替换成指定内容
* String replaceAll(String,String) 将某个内容全部替换成指定内容，支持正则
* String repalceFirst(String,String) 将第一次出现的某个内容替换成指定的内容
* String substring(int) 从指定下标开始一直截取到字符串的最后
* String substring(int,int) 从下标x截取到下标y-1对应的元素
* String trim() 去除一个字符串的前后空格

5、和位置有关：

* char charAt(int) 得到指定下标位置对应的字符
* int indexOf(String) 得到指定内容第一次出现的下标
* int lastIndexOf(String) 得到指定内容最后一次出现的下标

## 抽象类必须要有抽象方法吗？

答：抽象类中不一定要包含抽象(abstrace)方法。也就是了，抽象中可以没有抽象(abstract)方法。反之，类中含有抽象方法，那么类必须声明为抽象类。

## 普通类和抽象类有哪些区别？

答：

1. 抽象类不能被实例
2. 抽象类不能有构造函数，抽象方法也不能被声明为静态
3. 抽象类可以有抽象方法
4. 抽象类的抽象方法必须被非抽象子类继承

## 抽象类能使用 final 修饰吗？

不能，抽象类中的抽象方法是未来继承之后重写方法，而用final修饰的类，无法被继承。

## 接口和抽象类有什么区别？

答：

1. 抽象类是被子类继承，接口是被类实现
2. 接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现
3. 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量
4. 接口是设计的结果 ，抽象类是重构的结果

## java 中 IO 流分为几种？

可以分4种。

1. 字节输入流（**InputStream**）
2. 字节输出流（**OutputStream**）
3. 字符输入流（**Reader**）
4. 字符输出流（**Writer**）

## BIO、NIO、AIO 有什么区别？

同步阻塞IO（JAVA BIO）：   
    同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。   
同步非阻塞IO(Java NIO) ：

同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问。   
异步阻塞IO（Java NIO）：    
   此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄（如果从UNP的角度看，select属于同步操作。因为select之后，进程还需要读写数据），从而提高系统的并发性！    
（Java AIO(NIO.2)）异步非阻塞IO:    
   在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

## 17. Files的常用方法都有哪些？

**创建：**  
createNewFile()在指定位置创建一个空文件，成功就返回true，如果已存在就不创建，然后返回false。  
mkdir() 在指定位置创建一个单级文件夹。  
mkdirs() 在指定位置创建一个多级文件夹。  
renameTo(File dest)如果目标文件与源文件是在同一个路径下，那么renameTo的作用是重命名， 如果目标文件与源文件不是在同一个路径下，那么renameTo的作用就是剪切，而且还不能操作文件夹。

**删除：**  
delete() 删除文件或者一个空文件夹，不能删除非空文件夹，马上删除文件，返回一个布尔值。  
deleteOnExit()jvm退出时删除文件或者文件夹，用于删除临时文件，无返回值。  
**判断：**  
exists() 文件或文件夹是否存在。  
isFile() 是否是一个文件，如果不存在，则始终为false。  
isDirectory() 是否是一个目录，如果不存在，则始终为false。  
isHidden() 是否是一个隐藏的文件或是否是隐藏的目录。  
isAbsolute() 测试此抽象路径名是否为绝对路径名。  
**获取：**  
getName() 获取文件或文件夹的名称，不包含上级路径。  
getAbsolutePath()获取文件的绝对路径，与文件是否存在没关系  
length() 获取文件的大小（字节数），如果文件不存在则返回0L，如果是文件夹也返回0L。  
getParent() 返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回null。  
lastModified()获取最后一次被修改的时间。

**文件夹相关：**  
static File[] listRoots()列出所有的根目录（Window中就是所有系统的盘符）  
list() 返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回null。  
listFiles() 返回目录下的文件或者目录对象（File类实例），包含隐藏文件。对于文件这样操作会返回null。  
list(FilenameFilter filter)返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。  
listFiles(FilenameFilter filter)返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

# 2. 容器（集合）

## 18. java 容器都有哪些？（容器指的是集合类）

## ****基本概念****

Java容器类类库的用途是“持有对象”，并将其划分为两个不同的概念：

**１）Collection：一个独立元素的序列，这些元素都服从一条或者多条规则。** List必须按照插入的顺序保存元素，而set不能有重复的元素。Queue按照排队规则来确定对象产生的顺序（通常与它们被插入的顺序相同）。   
**2）Map：一组成对的“键值对”对象，允许你使用键来查找值。**

**|Collection**   
|　　├List   
|　　│-├LinkedList   
|　　│-├ArrayList   
|　　│-└Vector   
|　　│　└Stack   
|　　├Set   
|　　│├HashSet   
|　　│├TreeSet   
|　　│└LinkedSet   
|   
|**Map**   
　　├Hashtable   
　　├HashMap   
　　└WeakHashMap

## Collection 和 Collections 有什么区别？

注： **1、java.util.Collection 是一个集合接口。**它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式。   
　　**2、java.util.Collections 是一个包装类。**它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，服务于Java的Collection框架。

## List、Set、Map 之间的区别是什么？

答：

* List：
  + 可以允许重复对象
  + 可以插入多个null元素
  + 是一个有序容器
* Set：
  + 不允许重复对象
  + 只允许一个null元素
  + 无序容器
* Map:
  + Map不是Collection的子接口或实现类。Map是一个接口
  + Map 的每个Entry都特有两个对象，也就是一个键一个值，Map可能会持有相同的值对象但键对象必须是唯一的
  + Map里可以拥有随意个niull值但最多只能有一个null键

21.HashMap 和 Hashtable 有什么区别？  
 1.Map是一个以键值对存储的接口。Map下有两个具体的实现，分别是HashMap和HashTable.

2.HashMap是线程非安全的，HashTable是线程安全的，所以HashMap的效率高于HashTable.

3.HashMap允许键或值为空，而HashTable不允许键或值为空.

4.继承关系不同：

HashTable

public class Hashtable<K,V> extends Dictionary<K,V>1.0

implements Map<K,V>, Cloneable, java.io.Serializable {}

HashMap

public class HashMap<K,V> extends AbstractMap<K,V>1.2

implements Map<K,V>, Cloneable, Serializable{}

5.HashMap性能优于Hashtable。

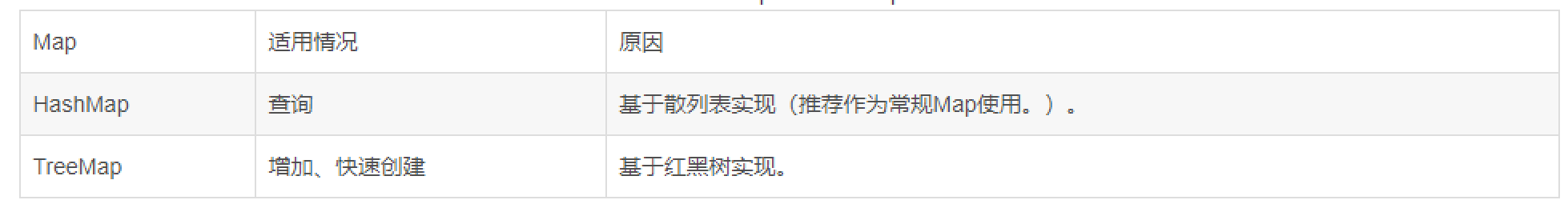
HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

### 附加问题：

#### 我们可以使用ConcurrentHashMap[1.5]来代替HashTable吗

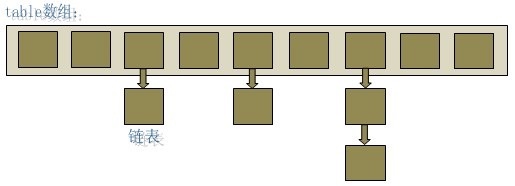
我们知道HashTable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁(分段式锁)。可以拿单例模式的双层锁模式来进行对比

## 22.如何决定使用 HashMap 还是 TreeMap？



## 说一下 HashMap 的实现原理？

**以下基于 JDK1.7 分析。**



如图所示，HashMap 底层是基于数组和链表实现的。其中有两个重要的参数：

容量

负载因子

容量的默认大小是 16，负载因子是 0.75，当 HashMap 的 size > 16\*0.75 时就会发生扩容(容量和负载因子都可以自由调整)。

**[put 方法](https://crossoverjie.top/JCSprout/" \l "/collections/HashMap?id=put-%e6%96%b9%e6%b3%95)**

首先会将传入的 Key 做 hash 运算计算出 hashcode,然后根据数组长度取模计算出在数组中的 index 下标。

由于在计算中位运算比取模运算效率高的多，所以 HashMap 规定数组的长度为 2^n 。这样用 2^n - 1 做位运算与取模效果一致，并且效率还要高出许多。

由于数组的长度有限，所以难免会出现不同的 Key 通过运算得到的 index 相同，这种情况可以利用链表来解决，HashMap 会在 table[index]处形成链表，采用头插法将数据插入到链表中。

**[get 方法](https://crossoverjie.top/JCSprout/" \l "/collections/HashMap?id=get-%e6%96%b9%e6%b3%95)**

get 和 put 类似，也是将传入的 Key 计算出 index ，如果该位置上是一个链表就需要遍历整个链表，通过 key.equals(k) 来找到对应的元素。

**[遍历方式](https://crossoverjie.top/JCSprout/" \l "/collections/HashMap?id=%e9%81%8d%e5%8e%86%e6%96%b9%e5%bc%8f)**

Iterator<Map.Entry<String, Integer>> entryIterator = map.entrySet().iterator();

while (entryIterator.hasNext()) {

Map.Entry<String, Integer> next = entryIterator.next();

System.out.println("key=" + next.getKey() + " value=" + next.getValue());

}

Iterator<String> iterator = map.keySet().iterator();

while (iterator.hasNext()){

String key = iterator.next();

System.out.println("key=" + key + " value=" + map.get(key));

}

map.forEach((key,value)->{

System.out.println("key=" + key + " value=" + value);});

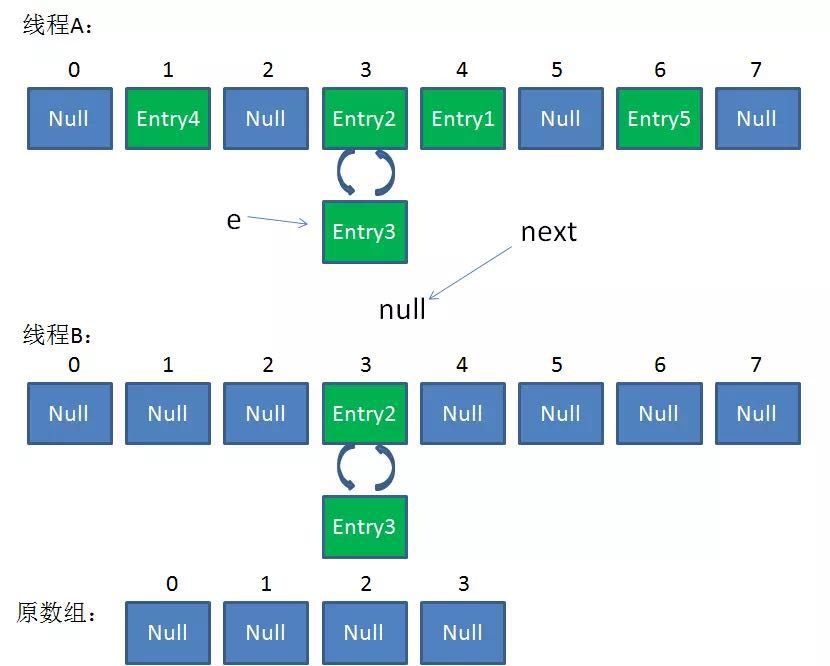
强烈建议使用第一种 EntrySet 进行遍历。

第一种可以把 key value 同时取出，第二种还得需要通过 key 取一次 value，效率较低, 第三种需要 JDK1.8 以上，通过外层遍历 table，内层遍历链表或红黑树。

**[notice](https://crossoverjie.top/JCSprout/" \l "/collections/HashMap?id=notice)**

在并发环境下使用 HashMap 容易出现死循环。

并发场景发生扩容，调用 resize() 方法里的 rehash() 时，容易出现环形链表。这样当获取一个不存在的 key时，计算出的 index 正好是环形链表的下标时就会出现死循环。



所以 HashMap 只能在单线程中使用，并且尽量的预设容量，尽可能的减少扩容。

在 JDK1.8 中对 HashMap 进行了优化： 当 hash 碰撞之后写入链表的长度超过了阈值(默认为8)并且 table 的长度不小于64(否则扩容一次)时，链表将会转换为红黑树。

假设 hash 冲突非常严重，一个数组后面接了很长的链表，此时重新的时间复杂度就是 O(n) 。

如果是红黑树，时间复杂度就是 O(logn) 。

大大提高了查询效率。

多线程场景下推荐使用 **[ConcurrentHashMap](https://github.com/crossoverJie/Java-Interview/blob/master/MD/ConcurrentHashMap.md" \t "https://crossoverjie.top/JCSprout/" \l "/collections/_blank)**。

## 说一下 HashSet 的实现原理？

HashSet 是一个不允许存储重复元素的集合，它的实现比较简单，只要理解了 HashMap，HashSet 就水到渠成了。

**[成员变量](https://crossoverjie.top/JCSprout/" \l "/collections/HashSet?id=%e6%88%90%e5%91%98%e5%8f%98%e9%87%8f)**

首先了解下 HashSet 的成员变量:

private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

发现主要就两个变量:

map ：用于存放最终数据的。

PRESENT ：是所有写入 map 的 value 值。

**[构造函数](https://crossoverjie.top/JCSprout/" \l "/collections/HashSet?id=%e6%9e%84%e9%80%a0%e5%87%bd%e6%95%b0)**

public HashSet() {

map = new HashMap<>();

}

public HashSet(int initialCapacity, float loadFactor) {

map = new HashMap<>(initialCapacity, loadFactor);

}

构造函数很简单，利用了 HashMap 初始化了 map 。

**[add](https://crossoverjie.top/JCSprout/" \l "/collections/HashSet?id=add)**

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

比较关键的就是这个 add() 方法。 可以看出它是将存放的对象当做了 HashMap 的健，value 都是相同的 PRESENT 。由于 HashMap 的 key 是不能重复的，所以每当有重复的值写入到 HashSet 时，value 会被覆盖，但 key 不会受到影响，这样就保证了 HashSet 中只能存放不重复的元素。

**[总结](https://crossoverjie.top/JCSprout/" \l "/collections/HashSet?id=%e6%80%bb%e7%bb%93)**

HashSet 的原理比较简单，几乎全部借助于 HashMap 来实现的。

所以 HashMap 会出现的问题 HashSet 依然不能避免。

## ArrayList 和 LinkedList 的区别是什么？

Arraylist：底层是基于动态数组，根据下表随机访问数组元素的效率高，向数组尾部添加元素的效率高；但是，删除数组中的数据以及向数组中间添加数据效率低，因为需要移动数组。例如最坏的情况是删除第一个数组元素，则需要将第2至第n个数组元素各向前移动一位。而之所以称为动态数组，是因为Arraylist在数组元素超过其容量大，Arraylist可以进行扩容（针对JDK1.8  数组扩容后的容量是扩容前的1.5倍），Arraylist源码中最大的数组容量是Integer.MAX\_VALUE-8，对于空出的8位，目前解释是 ：①存储Headerwords；②避免一些机器内存溢出，减少出错几率，所以少分配③最大还是能支持到Integer.MAX\_VALUE（当Integer.MAX\_VALUE-8依旧无法满足需求时）.

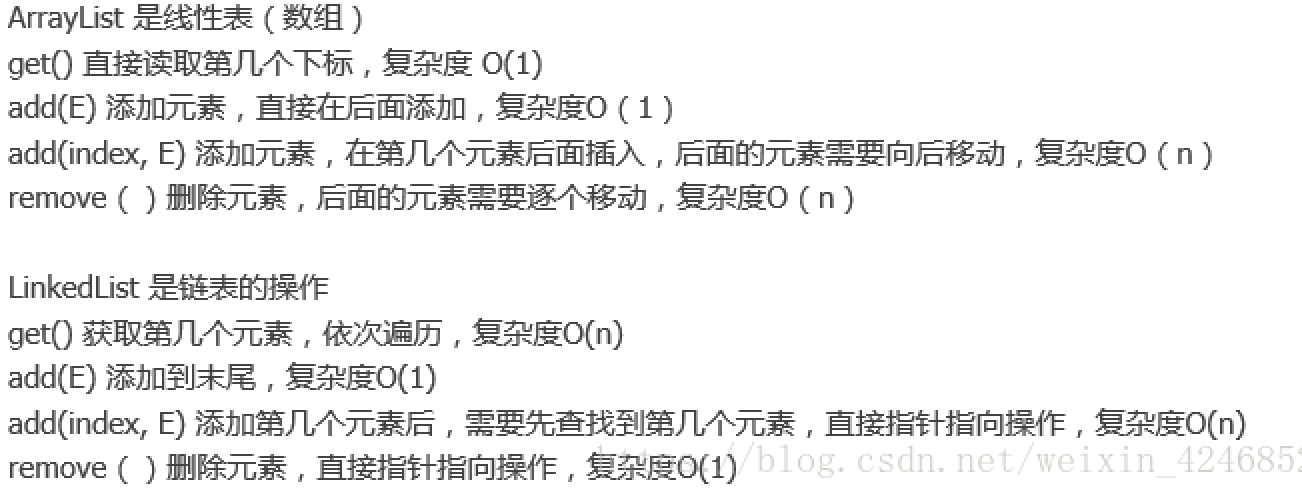
只要ArrayList的当前容足够大，add()操作向数组的尾部的效率非常高的，当向数组指定位置添加yi据时，会进行大量的数组移动复制操作。而数组复制时，最终将调用System.arraycopy()方法，因此add()操作的效率还是相当高的。尽管这样当向指定位置添加数据时也还是比Linkedlist慢，后者添加数据只需要改变指针指向即可。Arraylist删除数组也需要移动数组，效率较慢。

Linkedlist基于链表的动态数组，数据添加删除效率高，只需要改变指针指向即可，但是访问数据的平均效率低，需要对链表进行遍历。

总结：

1、对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。 对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

2、各自效率问题：



## 如何实现数组和 List 之间的转换？

List转数组：toArray(arraylist.size()方法

数组转List：

1.Arrays的asList(a)方法（效率不高）

2.通过集合工具类Collections.addAll()方法(最高效)

通过Collections.addAll(arrayList, strArray)方式转换，根据数组的长度创建一个长度相同的List，然后通过Collections.addAll()方法，将数组中的元素转为二进制，然后添加到List中，这是最高效的方法。

## ArrayList 和 Vector 的区别是什么？

ArrayList 实现于 List、RandomAccess 接口。可以插入空数据，也支持随机访问。

ArrayList相当于动态数据，其中最重要的两个属性分别是: elementData 数组，以及 size 大小。 在调用 add() 方法的时候：

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

首先进行扩容校验。

将插入的值放到尾部，并将 size + 1 。

如果是调用 add(index,e) 在指定位置添加的话：

public void add(int index, E element) {

rangeCheckForAdd(index);

ensureCapacityInternal(size + 1); // Increments modCount!!

//复制，向后移动

System.arraycopy(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

也是首先扩容校验。

接着对数据进行复制，目的是把 index 位置空出来放本次插入的数据，并将后面的数据向后移动一个位置。

其实扩容最终调用的代码:

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

也是一个数组复制的过程。

由此可见 ArrayList 的主要消耗是数组扩容以及在指定位置添加数据，在日常使用时最好是指定大小，尽量减少扩容。更要减少在指定位置插入数据的操作。

**[序列化](https://crossoverjie.top/JCSprout/" \l "/collections/ArrayList?id=%e5%ba%8f%e5%88%97%e5%8c%96)**

由于 ArrayList 是基于动态数组实现的，所以并不是所有的空间都被使用。因此使用了 transient 修饰，可以防止被自动序列化。

transient Object[] elementData;

因此 ArrayList 自定义了序列化与反序列化：

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException{

// Write out element count, and any hidden stuff

int expectedModCount = modCount;

s.defaultWriteObject();

// Write out size as capacity for behavioural compatibility with clone()

s.writeInt(size);

// Write out all elements in the proper order.

//只序列化了被使用的数据

for (int i=0; i<size; i++) {

s.writeObject(elementData[i]);

}

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

}

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

elementData = EMPTY\_ELEMENTDATA;

// Read in size, and any hidden stuff

s.defaultReadObject();

// Read in capacity

s.readInt(); // ignored

if (size > 0) {

// be like clone(), allocate array based upon size not capacity

ensureCapacityInternal(size);

Object[] a = elementData;

// Read in all elements in the proper order.

for (int i=0; i<size; i++) {

a[i] = s.readObject();

}

}

}

当对象中自定义了 writeObject 和 readObject 方法时，JVM 会调用这两个自定义方法来实现序列化与反序列化。

从实现中可以看出 ArrayList 只序列化了被使用的数据。

**[Vector](https://crossoverjie.top/JCSprout/" \l "/collections/ArrayList?id=vector)**

Vector 也是实现于 List 接口，底层数据结构和 ArrayList 类似,也是一个动态数组存放数据。不过是在 add() 方法的时候使用 synchronized 进行同步写数据，但是开销较大，所以 Vector 是一个同步容器并不是一个并发容器。

以下是 add() 方法：

public synchronized boolean add(E e) {

modCount++;

ensureCapacityHelper(elementCount + 1);

elementData[elementCount++] = e;

return true;

}

以及指定位置插入数据:

public void add(int index, E element) {

insertElementAt(element, index);

}

public synchronized void insertElementAt(E obj, int index) {

modCount++;

if (index > elementCount) {

throw new ArrayIndexOutOfBoundsException(index

+ " > " + elementCount);

}

ensureCapacityHelper(elementCount + 1);

System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);

elementData[index] = obj;

elementCount++;

}

## Array 和 ArrayList 有何区别？

Array可以包含基本类型和对象类型，ArrayList只能包含对象类型

　　Array大小固定，ArrayList的大小是动态变化的。

　　ArrayList提供了更多的方法和特性：比如 ：addAll(),removeAll(),iterator()等等。

　　对于基本数据类型，集合使用自动装箱来减少编码工作量。但是，当处理固定大小基本数据类型的时候，这种方式相对较慢。

## 29. 在Queue中poll()和remove()有什么区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

## 30. 哪些集合类是线程安全的？

Vector：就比ArrayList多了一个同步化机制（线程安全）

LinkedList因为成员方法大多是synchronized的，因此LinkedList是线程安全的。而ArrayList不是线程安全的。在扩容机制上，当Vector的元素数量超过它的初始化大小的时候会将容量翻倍，而ArrayList只会增长50%。

ArrayList的数据结构是基于数组的（Object[]）,而LinkList内部结构是基于一组链接的记录，形式上属于链表的。所以在增加元素方面linkList的效率更高，因为在ArrayList中增加元素，会牵扯一次重新排序。删除也是类似，所以ArrayList的查询性能要好些。反之LinkList增加，删除性能更好。如果是迭代读取的话，就没有什么差别了。

HashTable：比hashMap多了一个线程安全。hashTable的方法都提供了同步机制。hashTable不允许插入空值，hashMap是允许的。

ConcurrentHashMap：是一种高效但是也线程安全的集合。它比Hashmap来讲，是线程安全的，但是效率也比较高，因为它引入了一个分段锁的概念，可以理解为把一个大的Map拆分成了N个小的hashTable。根据key.hashCode（）决定把key放到哪个hashtable中。HashMap的数据结构是数据和链表。通过hash算法计算该key所在的数组下标，根据equals取比较值。通俗的说救赎ConcurrenthashMap是对每个数组进行加锁，当通过hash算法得出的结果相同时才需要去同步数据。

## 31. 迭代器 Iterator 是什么？

## 对 Collection 进行迭代的类，称其为迭代器。还是面向对象的思想，专业对象做专业的事情，迭代器就是专门取出集合元素的对象。但是该对象比较特殊，不能直接创建对象（通过new），该对象是以内部类的形式存在于每个集合类的内部。

## 32. Iterator 怎么使用？有什么特点？

Collection接口中定义了获取集合类迭代器的方法（iterator（）），所以所有的Collection体系集合都可以获取自身的迭代器。



## 33. Iterator 和 ListIterator 有什么区别？

1. ListIterator有add()方法，可以向List中添加对象，而Iterator不能

2. ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。

3. ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

4. 都可实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iierator仅能遍历，不能修改。

## 34. 怎么确保一个集合不能被修改？

Collections.unmodifiableList(List)  
Collections.unmodifiableMap(Map)  
Collections.unmodifiableSet(Set)

可以返回一个只读视图 不能修改

# 3. 多线程

## 35. 并行和并发有什么区别？

答案一：

****并发的关键是你有处理多个任务的能力，不一定要同时。****

并行的关键是你有同时处理多个任务的能力。

所以我认为它们最关键的点就是：是否是『同时』。

答案二：

**并发**（concurrency）和**并行**（parallellism）是：

1. 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
2. 解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件**。**
3. 解释三：在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

## 36. 线程和进程的区别？

答：

* 线程：是程序执行流的最小单元，是系统独立调度和分配CPU（独立运行）的基本单位
* 进程：是资源分配的基本单位。一个进程包括多个线程

区别：地址空间、资源拥有

1. 线程与资源分配无关，它属于某一个进程，并与进程内的其他线程一起共享进程的资源
2. 每个进程都有自己一套独立的资源（数据），供其内的所有线程共享
3. 不论是大小，开销线程要更“轻量级”
4. 一个进程内的线程通信比进程之间的通信更快速，有效。（因为共享变量）

## 37. 守护线程是什么？

答：守护线程是个服务线程，服务于其他线程

典型案例：垃圾回收线程

守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程，这是它的作用——而其他的线程只有一种，那就是用户线程。所以java里线程分2种，

1、守护线程，比如垃圾回收线程，就是最典型的守护线程。

2、用户线程，就是应用程序里的自定义线程。

## 38. 创建线程有哪几种方式？

1.继承 Thread 类，然后调用 start 方法。

2.实现 Runnable 接口的 run 方法, 然后再用 Thread 类包裹后，调用 start 方法。

3.实现 Callable 接口的 call 方法，用 FutureTask 类包裹 Callable 对象。然后再用 Thread 类包裹 FutureTask 类，并调用 start 方法。call() 方法可以有返回值。

注：Callable 方法在 Java 8 后，支持拉姆达表达式的写法，可以创建一个 FutureTask 类，语句上不是太罗嗦。 Callable 方式有以下几个优点：

* 可以捕获线程上的异常。
* 可以通过 get 方法得到返回值。
* get 方法阻塞当前线程，直到调用的线程运行结束。
* 可以取消线程的运行。

4.Java 6 之后，还可以通过创建线程池来创建线程，使用 ExecutorService 的 execute 方法:

## 39. 说一下 runnable 和 callable 有什么区别？

相同点：

1. 两者都是接口；（废话）
2. 两者都可用来编写多线程程序；
3. 两者都需要调用Thread.start()启动线程；

不同点：

1. 两者最大的不同点是：实现Callable接口的任务线程能返回执行结果；而实现Runnable接口的任务线程不能返回结果；
2. Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛；

注意点：

* Callable接口支持返回执行结果，此时需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取‘将来’结果；当不调用此方法时，主线程不会阻塞！

## 40. 线程有哪些状态？

## 线程状态有 5 种，新建，就绪，运行，阻塞，死亡。

1. 线程 start 方法执行后，并不表示该线程运行了，而是进入就绪状态，意思是随时准备运行，但是真正何时运行，是由操作系统决定的，代码并不能控制，

2. 同样的，从运行状态的线程，也可能由于失去了 CPU 资源，回到就绪状态，也是由操作系统决定的。这一步中，也可以由程序主动失去 CPU 资源，只需调用 yield 方法。

3. 线程运行完毕，或者运行了一半异常了，或者主动调用线程的 stop 方法，那么就进入死亡。死亡的线程不可逆转。

4. 下面几个行为，会引起线程阻塞。

* 主动调用 sleep 方法。时间到了会进入就绪状态
* 主动调用 suspend 方法。主动调用 resume 方法，会进入就绪状态
* 调用了阻塞式 IO 方法。调用完成后，会进入就绪状态。
* 试图获取锁。成功的获取锁之后，会进入就绪状态。
* 线程在等待某个通知。其它线程发出通知后，会进入就绪状态

## 41. sleep() 和 wait() 有什么区别？

一：1、这两个方法来自不同的类分别是Thread和Object，sleep方法属于Thread类中的静态方法，wait属于Object的成员方法。

2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用（使用范围）。

二：1..sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。

在调用sleep()方法的过程中，线程不会释放对象锁。

2.一旦一个对象调用了wait方法，必须要采用notify()和notifyAll()方法唤醒该线程;如果线程拥有某个或某些对象的同步锁，那么在调用了wait()后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了wait()方法的对象。

## 42. notify()和 notifyAll()有什么区别？

notify():  
唤醒一个处于等待状态的线程，  
注意的是在调用此方法的时候，  
并不能确切的唤醒某一个等待状态的线程，  
而是由JVM确定唤醒哪个线程，而且不是按优先级。

notifyAll():  
唤醒所有处入等待状态的线程;  
并可以理解为把他们排进一个队列;  
只不过只有头部的线程获得了锁，才能运行;  
注意！！并不是给所有唤醒线程一个对象的锁，而是让它们竞争，  
当其中一个线程运行完就开始运行下一个已经被唤醒的线程，因为锁已经转移了。  
（这个时候是否运行已经不是因为等待状态，而是处于runnning队列中）

## 43. 线程的 run()和 start()有什么区别？

## 1.start（）方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。 2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

## 44.创建线程池有哪几种方式？

### 1.ThreadPoolExecutor，

public class Main {

public static void main(String[] args) throws Exception {

// 有界队列

BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(5);

// 放弃拒绝的任务并抛出异常

RejectedExecutionHandler abortPolicyHandler = new ThreadPoolExecutor.AbortPolicy();

RejectedExecutionHandler discardPolicyHandler = new ThreadPoolExecutor.DiscardPolicy();

ThreadPoolExecutor threadPool =

new ThreadPoolExecutor(5, 10, 30, TimeUnit.SECONDS, workQueue, discardPolicyHandler);

long start = System.currentTimeMillis();

for (int i = 0; i < 40; i++) {

threadPool.execute(new MyTask());

System.out.println("核心线程数" + threadPool.getCorePoolSize());

System.out.println("最大线程数" + threadPool.getMaximumPoolSize());

System.out.println("线程池数" + threadPool.getPoolSize());

System.out.println("队列任务数" + threadPool.getQueue().size());

System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");

}

System.out.println(System.currentTimeMillis()-start);

threadPool.shutdown();

if (threadPool.awaitTermination(6, TimeUnit.SECONDS)) {

threadPool.shutdownNow();

}

}

}

class MyTask implements Runnable {

@Override

public void run() {

try {

Thread.currentThread().sleep(5000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + ", hello");

}

}

### 2.Executors

因为ThreadPoolExecutor线程池类创建线程较为麻烦，需要设置的参数比较多。线程池工具类，提供了一些工厂方法或者实用方法用来创建或者使用线程池，使我们更加方便快捷的创建和使用线程池。

/\*

\* Java通过Executors提供四种线程池，分别为：

\* newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

\* newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

\* newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

\* newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

\*/

## 45.线程池都有哪些状态？

## 线程池的5种状态：Running、ShutDown、Stop、Tidying、Terminated。

1、RUNNING

(1) 状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。   
(02) 状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0！

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

2、 SHUTDOWN

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

3、STOP

(1) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。   
(2) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

4、TIDYING

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

5、 TERMINATED

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

## 46. 线程池中 submit()和 execute()方法有什么区别？

* execute()：只能执行 Runnable 类型的任务。
* submit()：可以执行 Runnable 和 Callable 类型的任务。

Callable 类型的任务可以获取执行的返回值，而 Runnable 执行无返回值。

## 在 java 程序中怎么保证多线程的运行安全？

答：

* 方法一：使用安全类，比如 Java. util. concurrent 下的类。
* 方法二：使用自动锁 synchronized。
* 方法三：使用手动锁 Lock。

手动锁Java示例代码如下：

Lock lock = new ReentrantLock();

lock.lock();try {

System. out. println("获得锁");

} catch (Exception e) {

// TODO: handle exception

} finally {

System. out. println("释放锁");

lock. unlock();

}

## 多线程锁的升级原理是什么？

答：

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

## 什么是死锁？

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

## 50. 怎么防止死锁？

答：

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

## 51. ThreadLocal 是什么？有哪些使用场景？

## 答：ThreadLocal用于保存某个线程共享变量。使用场景：解决数据库连接，Session管理

## 52. 说一下 synchronized 底层实现原理？

## 答：synchronized 是由一对 monitorenter/monitorexit 指令实现的，monitor 对象是同步的基本实现单元。在 Java 6 之前，monitor 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作，性能也很低。但在 Java 6 的时候，Java 虚拟机 对此进行了大刀阔斧地改进，提供了三种不同的 monitor 实现，也就是常说的三种不同的锁：偏向锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

## 53. synchronized 和 volatile 的区别是什么？

答：

* volatile 是变量修饰符；synchronized 是修饰类、方法、代码段。
* volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
* volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

## 54. synchronized 和 Lock 有什么区别？

* synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。
* synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。
* 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

## 55. synchronized 和 ReentrantLock 区别是什么？

答：

synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

主要区别如下：

* ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；
* ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；
* ReentrantLock 只适用于代码块锁，而 synchronized 可用于修饰方法、代码块等。
* volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

## 56. 说一下 atomic 的原理？

答：atomic 主要利用 CAS (Compare And Wwap) 和 volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

# 4. 反射

## 57. 什么是反射？

## 答：可以将一个程序（类）在运行的时候获得该程序（类）的信息的机制，也就是获得在编译期不可能获得的类的信息，因为这些信息是保存在Class对象中的，而这个Class对象是在程序运行时动态加载的

## 58. 什么是 java 序列化？什么情况下需要序列化？

答：系列化就是把java对象转换为字节序列的方法。

* 把对象的字节序列化到永久的保存到硬盘中
* 在网络上传递对象的字节序列

## 59. 动态代理是什么？有哪些应用？

答：动态代理是运行时动态生成代理类。

* 动态代理指的是可以任意控制任意对象的执行过程
  + 本来应该[自己做](https://www.baidu.com/s?wd=%E8%87%AA%E5%B7%B1%E5%81%9A&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的事情，因为没有某种原因不能直接做，只能请别人代理做。被请的人就是代理
  + 比如春节买票回家，由于没有时间，只能找票务中介来买，这就是代理模式
* 应用：Spring 的 AOP

## 60. 怎么实现动态代理？

答：

* JDK动态代理
* cglib动态代理

# 5. 对象拷贝

## 61. 为什么要使用克隆？

答：克隆的对象可能包含一些已经修改过的属性，而 new 出来的对象的属性都还是初始化时候的值，所以当需要一个新的对象来保存当前对象的“状态”就靠克隆方法了。

## 如何实现对象克隆？

答：

* 实现 Cloneable 接口并重写 Object 类中的 clone() 方法。
* 实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆。

## 63. 深拷贝和浅拷贝区别是什么？

答：

* 浅克隆：当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制。
* 深克隆：除了对象本身被复制外，对象所包含的所有成员变量也将复制。

# 6. Java Web

## 64. jsp 和 servlet 有什么区别？

答：

* jsp经编译后就成了Servlet（Jsp的本质就是Servlet，JVM只能识别Java类，不能识别Jsp代码，Web容器将Jsp的代码编译成JVM能够识别的java类）
* jsp更擅长表现于页面显示，servlet更擅长于逻辑控制
* Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到

## 65. jsp 有哪些内置对象？作用分别是什么？

## 说一下 jsp 的 4 种作用域？

## 67. session 和 cookie 有什么区别？

答：

* 存储位置不同：session 存储在服务器端；cookie 存储在浏览器端。
* 安全性不同：cookie 安全性一般，在浏览器存储，可以被伪造和修改。
* 容量和个数限制：cookie 有容量限制，每个站点下的 cookie 也有个数限制。
* 存储的多样性：session 可以存储在 Redis 中、数据库中、应用程序中；而 cookie 只能存储在浏览器中

## 68. 说一下 session 的工作原理？

## 答：session 的工作原理是客户端登录完成之后，服务器会创建对应的 session，session 创建完之后，会把 session 的 id 发送给客户端，客户端再存储到浏览器中。这样客户端每次访问服务器时，都会带着 sessionid，服务器拿到 sessionid 之后，在内存找到与之对应的 session 这样就可以正常工作了

## 69. 如果客户端禁止 cookie 能实现 session 还能用吗？

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

## 70. spring mvc 和 struts 的区别是什么？

答：

* 拦截级别：struts2 是类级别的拦截；spring mvc 是方法级别的拦截。
* 数据独立性：spring mvc 的方法之间基本上独立的，独享 request 和 response 数据，请求数据通过参数获取，处理结果通过 ModelMap 交回给框架，方法之间不共享变量；而 struts2 虽然方法之间也是独立的，但其所有 action 变量是共享的，这不会影响程序运行，却给我们编码和读程序时带来了一定的麻烦。
* 拦截机制：struts2 有以自己的 interceptor 机制，spring mvc 用的是独立的 aop 方式，这样导致struts2 的配置文件量比 spring mvc 大。
* 对 ajax 的支持：spring mvc 集成了ajax，所有 ajax 使用很方便，只需要一个注解 @ResponseBody 就可以实现了；而 struts2 一般需要安装插件或者自己写代码才行。

## 71. 如何避免 sql 注入？

答：

* 使用预处理 PreparedStatement。
* 使用正则表达式过滤掉字符中的特殊字符。

## 72. 什么是 XSS 攻击，如何避免？

答：

XSS 攻击：即跨站脚本攻击，它是 Web 程序中常见的漏洞。原理是攻击者往 Web 页面里插入恶意的脚本代码（css 代码、Javascript 代码等），当用户浏览该页面时，嵌入其中的脚本代码会被执行，从而达到恶意攻击用户的目的，如盗取用户 cookie、破坏页面结构、重定向到其他网站等。

预防 XSS 的核心是必须对输入的数据做过滤处理。

## 73. 什么是 CSRF 攻击，如何避免？

答：

CSRF：Cross-Site Request Forgery（中文：跨站请求伪造），可以理解为攻击者盗用了你的身份，以你的名义发送恶意请求，比如：以你名义发送邮件、发消息、购买商品，虚拟货币转账等。

防御手段：

* 验证请求来源地址；
* 关键操作添加验证码；
* 在请求地址添加 token 并验证

# 7. 异常

## 74. throw 和 throws 的区别？

答：

* throw则是指抛出的一个具体异常类型
* throws是用来声明一个方法可能抛出的所有异常信息

## 75. final、finally、finalize 有什么区别？

答：

* final 是用来修饰类、方法、变量
* finally 只能用在 try catch 语法中，表示这段语句最终一定会被执行

## 76. try-catch-finally 中哪个部分可以省略？

## 答：try-catch-finally 其中 catch 和 finally 都可以被省略，但是不能同时省略，也就是说有 try 的时候，必须后面跟一个 catch 或者 finally

## 77. try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？

## 答：一定会，catch 中 return会等finally中的代码执行完之后才会执行

## 78. 常见的异常类有哪些？

* NullPointerException 空指针异常
* ClassNotFoundException 指定类不存在
* NumberFormatException 字符串转换为数字异常
* IndexOutOfBoundsException 数组下标越界异常
* ClassCastException 数据类型转换异常
* FileNotFoundException 文件未找到异常
* NoSuchMethodException 方法不存在异常
* IOException IO 异常
* SocketException Socket 异常

# 8. 网络

## 79. http 响应码 301 和 302 代表的是什么？有什么区别？

答：

* 301表示网页永久性转移到另一个地址
* 302表示临时性转移
* 区别：
  + 301是永久的重定向，搜索引擎在抓取新内容的同时也将旧的网址替换为重定向之后的网址
  + 302重定向是临时的重定向，搜索引擎抓取新的内容而保留旧的网址

## 80. forward 和 redirect 的区别？

答：

* forward 是服务器的内部重定向
* redirect 是服务器收到请求后发送一个状态头给客户，客户将在重新请求一次

## 81. 简述 tcp 和 udp的区别？

答：

tcp 和 udp 是 OSI 模型中的运输层中的协议。tcp 提供可靠的通信传输，而 udp 则常被用于让广播和细节控制交给应用的通信传输。

两者的区别大致如下：

* tcp 面向连接，udp 面向非连接即发送数据前不需要建立链接；
* tcp 提供可靠的服务（数据传输），udp 无法保证；
* tcp 面向字节流，udp 面向报文；
* tcp 数据传输慢，udp 数据传输快

## 82. tcp 为什么要三次握手，两次不行吗？为什么？

答：

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

## 83. 说一下 tcp 粘包是怎么产生的？

答：

tcp 粘包可能发生在发送端或者接收端，分别来看两端各种产生粘包的原因：

* 发送端粘包：发送端需要等缓冲区满才发送出去，造成粘包；
* 接收方粘包：接收方不及时接收缓冲区的包，造成多个包接收。

## 84. OSI 的七层模型都有哪些？

答：

* 物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。
* 数据链路层：负责建立和管理节点间的链路。
* 网络层：通过路由选择算法，为报文或分组通过通信子网选择最适当的路径。
* 传输层：向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输。
* 会话层：向两个实体的表示层提供建立和使用连接的方法。
* 表示层：处理用户信息的表示问题，如编码、数据格式转换和加密解密等。
* 应用层：直接向用户提供服务，完成用户希望在网络上完成的各种工作。

## 85. get 和 post 请求有哪些区别？

答：

* get请求的参数在URL中，而post 请求在请求body中
* get请求传参有长度限制，post请求没有长度限制
* get请求的参数只能是ASCII码，post请求传参没有这个限制

## 86. 如何实现跨域？

答：

* 使用CORS技术
* 使用JSONP

## 说一下 JSONP 实现原理？

答：jsonp是一种轻量级的数据交换格式。

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

# 9. 设计模式

## 88. 说一下你熟悉的设计模式？

答：

* 单例模式：保证被创建一次，节省系统开销。
* 工厂模式（简单工厂、抽象工厂）：解耦代码。
* 观察者模式：定义了对象之间的一对多的依赖，这样一来，当一个对象改变时，它的所有的依赖者都会收到通知并自动更新。
* 外观模式：提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层的接口，让子系统更容易使用。
* 模版方法模式：定义了一个算法的骨架，而将一些步骤延迟到子类中，模版方法使得子类可以在不改变算法结构的情况下，重新定义算法的步骤。
* 状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

## 89. 简单工厂和抽象工厂有什么区别？

答：

* 简单工厂：用来生产同一等级结构中的任意产品，对于增加新的产品，无能为力。
* 工厂方法：用来生产同一等级结构中的固定产品，支持增加任意产品。
* 抽象工厂：用来生产不同产品族的全部产品，对于增加新的产品，无能为力；支持增加产品族

# 10. Spring/Spring MVC

## 90. 为什么要使用 spring？

答：spring是一个开源框架，是个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架

* 方便结构简化开发
* AOP编码的支持
* 声明式事物的支持
* 方便程序的测试
* 方便集成各种优势框架
* 降低Java EE API 的使用难度

## 解释一下什么是 aop？

## 解释一下什么是 ioc？

## 93. spring 有哪些主要模块？

答：core模块、aop模块、data access模块、web模块、test模块

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

## 94. spring 常用的注入方式有哪些？

答：详细解析：[https://blog.csdn.net/a909301740/article/details/78379720](https://blog.csdn.net/a909301740/article/details/78379720" \t "https://www.cnblogs.com/chen1005/p/_blank)

* 构造方法注入
* setter注入
* 基于注解注入

## 95. spring 中的 bean 是线程安全的吗？

答：

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 无状态的（比如 dao 类），所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。

* 有状态就是有数据存储功能。
* 无状态就是不会保存数据。

## 96. spring 支持几种 bean 的作用域？

答：singleton、prototype、request、session、globalSession五中作用域。

* singleton：spring ioc 容器中只存在一个 bean 实例，bean 以单例模式存在，是系统默认值；
* prototype：每次从容器调用 bean 时都会创建一个新的示例，既每次 getBean()相当于执行 new Bean()操作；
* Web 环境下的作用域：
  + request：每次 http 请求都会创建一个 bean；
  + session：同一个 http session 共享一个 bean 实例；
  + global-session：用于 portlet 容器，因为每个 portlet 有单独的 session，globalsession 提供一个全局性的 http session

## 97. spring 自动装配 bean 有哪些方式？

答：可分为四种：

* byName：按照bean的属性名称来匹配要装配的bean
* byType：按照bean的类型来匹配要装配的bean
* constructor：按照bean的构造器入参的类型来进行匹配
* autodetect（自动检测）：先使用constructor进行装配，如果不成功就使用byType来装配

## 98. spring 事务实现方式有哪些？

答：

* 声明式事务：声明式事务也有两种实现方式，基于 xml 配置文件的方式和注解方式（在类上添加 @Transaction 注解）。
* 编码方式：提供编码的形式管理和维护事务。

## 99. 说一下 spring 的事务隔离？

答：

spring 有五大隔离级别，默认值为 ISOLATION\_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

* ISOLATION\_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
* ISOLATIONREADUNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；
* ISOLATIONREADCOMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
* ISOLATIONREPEATABLEREAD：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
* ISOLATION\_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。
* 脏读 ：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。
* 不可重复读 ：是指在一个事务内，多次读同一数据。
* 幻读 ：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了

## 100. 说一下 spring mvc 运行流程？

答：

* spring mvc 先将请求发送给 DispatcherServlet。
* DispatcherServlet 查询一个或多个 HandlerMapping，找到处理请求的 Controller。
* DispatcherServlet 再把请求提交到对应的 Controller。
* Controller 进行业务逻辑处理后，会返回一个ModelAndView。
* Dispathcher 查询一个或多个 ViewResolver 视图解析器，找到 ModelAndView 对象指定的视图对象。
* 视图对象负责渲染返回给客户端

## 101. spring mvc 有哪些组件？

答：

* 前置控制器 DispatcherServlet
* 映射控制器 HandlerMapping
* 处理器 Controller
* 模型和视图 ModelAndView
* 视图解析器 ViewResolver

## 102. @RequestMapping 的作用是什么？

## 答：将 http 请求映射到相应的类/方法上

## 103. @Autowired 的作用是什么？

答：@Autowired 它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作，通过@Autowired 的使用来消除 set/get 方法

# 11. Spring Boot/Spring Cloud

## 104.什么是 spring boot？

答：Spring Boot是一个构建在Spring框架顶部的项目。它提供了一种更简单、更快捷的方法来设置、配置和运行简单和基于Web的应用程序。

## 105.为什么要用 spring boot？

答：

* 配置简单
* 独立运行
* 自动装配
* 无代码生成和 xml 配置
* 提供应用监控
* 易上手
* 提升开发效率

## 106. spring boot 核心配置文件是什么？

spring boot 核心的两个配置文件：

* bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，且 boostrap 里面的属性不能被覆盖
* application (. yml 或者 . properties)：用于 spring boot 项目的自动化配置

## 107. spring boot 配置文件有哪几种类型？它们有什么区别？

配置文件有 . properties 格式和 . yml 格式，它们主要的区别是书法风格不同。

. properties 配置如下：

spring. RabbitMQ. port=5672

. yml 配置如下：

spring:

RabbitMQ:

port: 5672

. yml 格式不支持 @PropertySource 注解导入

## 108. spring boot 有哪些方式可以实现热部署？

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

## 109. jpa 和 hibernate 有什么区别？

## jpa 全称 Java Persistence API，是 Java 持久化接口规范，hibernate 属于 jpa 的具体实现。

## 什么是 spring cloud？

spring cloud 是一系列框架的有序集合。它利用 spring boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 spring boot 的开发风格做到一键启动和部署

## 111. spring cloud 断路器的作用是什么？

## 在分布式架构中，断路器模式的作用也是类似的，当某个服务单元发生故障（类似用电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延

## spring cloud 的核心组件有哪些？

* Eureka：服务注册于发现
* Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求
* Ribbon：实现负载均衡，从一个服务的多台机器中选择一台
* Hystrix：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题
* Zuul：网关管理，由 Zuul 网关转发请求给对应的服务

# 12. Hibernate

113. 为什么要使用 hibernate？

114. 什么是 ORM 框架？

115. hibernate 中如何在控制台查看打印的 sql 语句？

116. hibernate 有几种查询方式？

117. hibernate 实体类可以被定义为 final 吗？

118. 在 hibernate 中使用 Integer 和 int 做映射有什么区别？

119. hibernate 是如何工作的？

120. get()和 load()的区别？

121. 说一下 hibernate 的缓存机制？

122. hibernate 对象有哪些状态？

123. 在 hibernate 中 getCurrentSession 和 openSession 的区别是什么？

124. hibernate 实体类必须要有无参构造函数吗？为什么？

# 13. Mybatis

125. mybatis 中 #{}和 ${}的区别是什么？

126. mybatis 有几种分页方式？

127. RowBounds 是一次性查询全部结果吗？为什么？

128. mybatis 逻辑分页和物理分页的区别是什么？

129. mybatis 是否支持延迟加载？延迟加载的原理是什么？

130. 说一下 mybatis 的一级缓存和二级缓存？

131. mybatis 和 hibernate 的区别有哪些？

132. mybatis 有哪些执行器（Executor）？

133. mybatis 分页插件的实现原理是什么？

134. mybatis 如何编写一个自定义插件？

# 14. RabbitMQ

135. rabbitmq 的使用场景有哪些？

136. rabbitmq 有哪些重要的角色？

137. rabbitmq 有哪些重要的组件？

138. rabbitmq 中 vhost 的作用是什么？

139. rabbitmq 的消息是怎么发送的？

140. rabbitmq 怎么保证消息的稳定性？

141.rabbitmq 怎么避免消息丢失？

142. 要保证消息持久化成功的条件有哪些？

143. rabbitmq 持久化有什么缺点？

144. rabbitmq 有几种广播类型？

145. rabbitmq 怎么实现延迟消息队列？

146. rabbitmq 集群有什么用？

147. rabbitmq 节点的类型有哪些？

148. rabbitmq 集群搭建需要注意哪些问题？

149. rabbitmq 每个节点是其他节点的完整拷贝吗？为什么？

150. rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况？

151. rabbitmq 对集群节点停止顺序有要求吗？

# 15. Kafka

152. kafka 可以脱离 zookeeper 单独使用吗？为什么？

153. kafka 有几种数据保留的策略？

154. kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？

155. 什么情况会导致 kafka 运行变慢？

156. 使用 kafka 集群需要注意什么？

# 16. Zookeeper

## 157. 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，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等

## 158. zookeeper 都有哪些功能？

## 159. zookeeper 有几种部署模式？

zookeeper的安装分为三种模式：单机模式、集群模式和伪集群模式。

单机模式

下载zookeeper的安装包之后, 解压到合适目录(/opt/modules/). 进入zookeeper目录下的conf子目录, 创建zoo.cfg

tickTime=2000

dataDir=/Users/apple/zookeeper/data

dataLogDir=/Users/apple/zookeeper/logs

clientPort=4180

参数说明:

 tickTime: zookeeper中使用的基本时间单位, 毫秒值.

 dataDir: 数据目录. 可以是任意目录.

dataLogDir: log目录, 同样可以是任意目录. 如果没有设置该参数, 将使用和dataDir相同的设置.

clientPort: 监听client连接的端口号.

至此, zookeeper的单机模式已经配置好了. 启动server只需运行脚本:

1. zkServer.sh start

 Server启动之后, 就可以启动client连接server了, 执行脚本:

1. zkCli.sh -server localhost:4180

## 伪集群模式

所谓伪集群，是指在单台机器中启动多个zookeeper进程，并组成一个集群，以启动3个zookeeper进程为例

将zookeeper的目录拷贝2份

更改zookeeper0/conf/zoo.cfg文件为：

TickTime=2000

initLimit=5

syncLimit=2

dataDir=/User/apple/zookeeper0/data

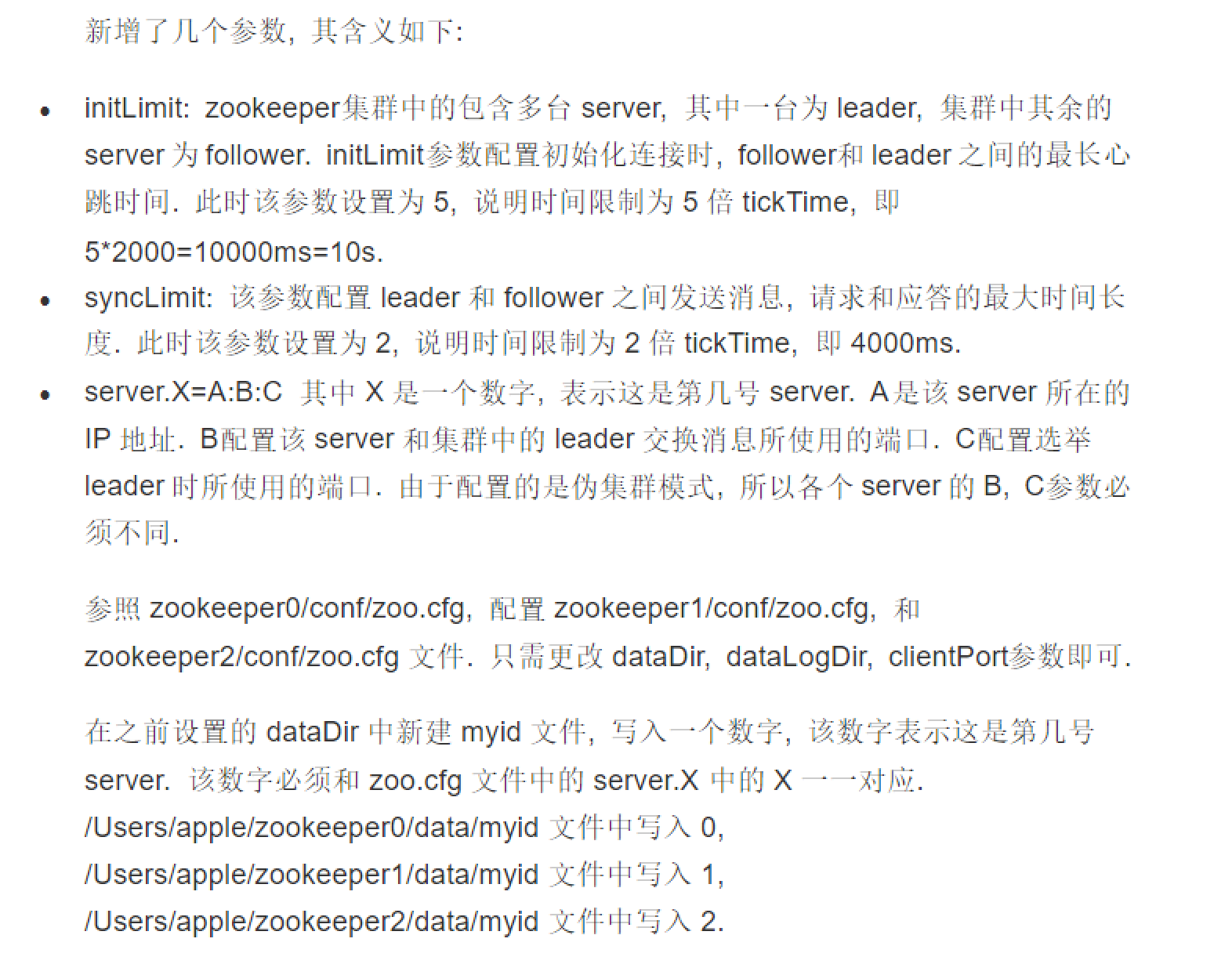
dataLogDir=/User/apple/zookeeper0/logs

ClientPort=4180

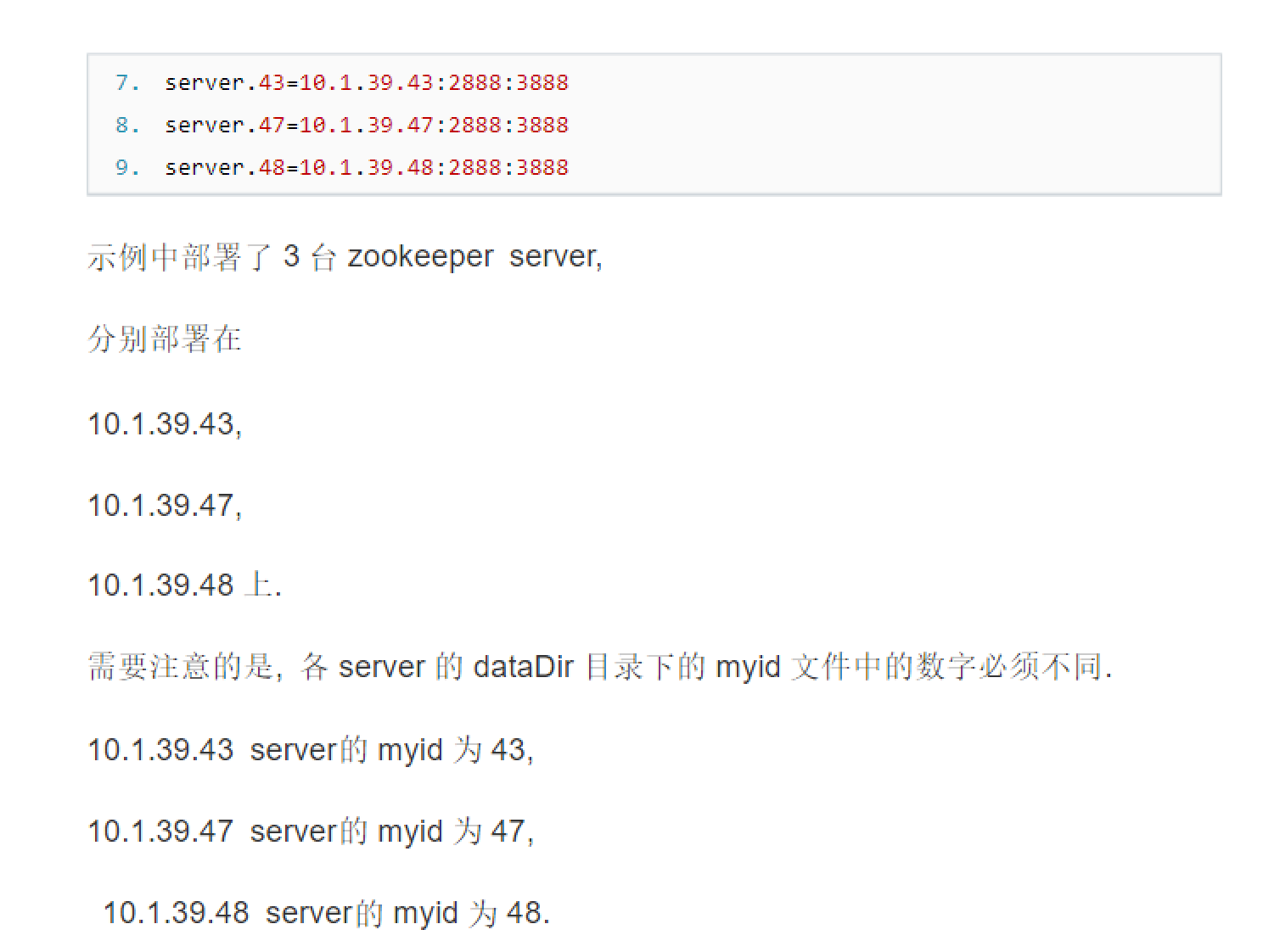
Server.0=127.0.0.1:8080:7770

Server.1=127.0.0.1:8081:7771

Server.2=127.0.0.1:8082:7772







## 160. zookeeper 怎么保证主从节点的状态同步？

## 161. 集群中为什么要有主节点？

## 162. 集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？

## 163. 说一下 zookeeper 的通知机制？

# 17. MySql

## 164. 数据库的三范式是什么？

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

## 165. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时 id 是几？

* 表类型如果是 MyISAM ，那 id 就是 8。
* 表类型如果是 InnoDB，那 id 就是 6。

## 166. 如何获取当前数据库版本？

使用 select version() 获取当前 MySQL 数据库版本。

select version()

## 167. 说一下 ACID 是什么？

* Atomicity（原子性）：一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。
* Consistency（一致性）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器、级联回滚等。
* Isolation（隔离性）：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
* Durability（持久性）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

## 168. char 和 varchar 的区别是什么？

* char(n) ：固定长度类型，比如订阅 char(10)，当你输入"abc"三个字符的时候，它们占的空间还是 10 个字节，其他 7 个是空字节。

chat 优点：效率高；缺点：占用空间；适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

* varchar(n) ：可变长度，存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度。

所以，从空间上考虑 varcahr 比较合适；从效率上考虑 char 比较合适，二者使用需要权衡

## 169. float 和 double 的区别是什么？

* float 最多可以存储 8 位的十进制数，并在内存中占 4 字节。
* double 最可可以存储 16 位的十进制数，并在内存中占 8 字节。

## 170. mysql 的内连接、左连接、右连接有什么区别？

## 内连接关键字：inner join；左连接：left join；右连接：right join。 内连接是把匹配的关联数据显示出来；左连接是左边的表全部显示出来，右边的表显示出符合条件的数据；右连接正好相反

## 171. mysql 索引是怎么实现的？

## 索引是满足某种特定查找算法的数据结构，而这些数据结构会以某种方式指向数据，从而实现高效查找数据。 具体来说 MySQL 中的索引，不同的数据引擎实现有所不同，但目前主流的数据库引擎的索引都是 B+ 树实现的，B+ 树的搜索效率，可以到达二分法的性能，找到数据区域之后就找到了完整的数据结构了，所有索引的性能也是更好的

## 172. 怎么验证 mysql 的索引是否满足需求？

使用 explain 查看 SQL 是如何执行查询语句的，从而分析你的索引是否满足需求。

explain 语法：

explain select \* from table where type=**1**

## 173. 说一下数据库的事务隔离？

MySQL 的事务隔离是在 MySQL. ini 配置文件里添加的，在文件的最后添加：

transaction-isolation = REPEATABLE-READ

可用的配置值：READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ、SERIALIZABLE。

* READ-UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）。
* READ-COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读）。
* REPEATABLE-READ：可重复读，默认级别，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读）。
* SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

脏读 ：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

不可重复读 ：是指在一个事务内，多次读同一数据。

幻读 ：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了

## 174. 说一下 mysql 常用的引擎？

* InnoDB 引擎：InnoDB 引擎提供了对数据库 acid 事务的支持，并且还提供了行级锁和外键的约束，它的设计的目标就是处理大数据容量的数据库系统。MySQL 运行的时候，InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎是不支持全文搜索，同时启动也比较的慢，它是不会保存表的行数的，所以当进行 select count(\*) from table 指令的时候，需要进行扫描全表。由于锁的粒度小，写操作是不会锁定全表的,所以在并发度较高的场景下使用会提升效率
* MyIASM 引擎：MySQL 的默认引擎，但不提供事务的支持，也不支持行级锁和外键。因此当执行插入和更新语句时，即执行写操作的时候需要锁定这个表，所以会导致效率会降低。不过和 InnoDB 不同的是，MyIASM 引擎是保存了表的行数，于是当进行 select count(\*) from table 语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的读操作远远多于写操作时，并且不需要事务的支持的，可以将 MyIASM 作为数据库引擎的首选

## 175. 说一下 mysql 的行锁和表锁？

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁

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

## 说一下乐观锁和悲观锁？

* 乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据
* 悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻止，直到这个锁被释放

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

## 177. mysql 问题排查都有哪些手段？

* 使用 show processlist 命令查看当前所有连接信息
* 使用 explain 命令查询 SQL 语句执行计划
* 开启慢查询日志，查看慢查询的 SQL

## 178. 如何做 mysql 的性能优化？

* 为搜索字段创建索引
* 避免使用 select \*，列出需要查询的字段
* 垂直分割分表
* 选择正确的存储引擎

# 18. Redis

179. redis 是什么？都有哪些使用场景？

180. redis 有哪些功能？

181. redis 和 memecache 有什么区别？

182. redis 为什么是单线程的？

183. 什么是缓存穿透？怎么解决？

184. redis 支持的数据类型有哪些？

185. redis 支持的 java 客户端都有哪些？

186. jedis 和 redisson 有哪些区别？

187. 怎么保证缓存和数据库数据的一致性？

188. redis 持久化有几种方式？

189.redis 怎么实现分布式锁？

190. redis 分布式锁有什么缺陷？

191. redis 如何做内存优化？

192. redis 淘汰策略有哪些？

193. redis 常见的性能问题有哪些？该如何解决？

# 19. JVM

## 194. 说一下 jvm 的主要组成部分？及其作用？

答：

* 类加载器（ClassLoader）
* 运行时数据区（Runtime Data Area）
* 执行引擎（Execution Engine）
* 本地库接口（Native Interface）

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

## 195. 说一下 jvm 运行时数据区？

不同虚拟机的运行时数据区可能略微有所不同，但都会遵从 Java 虚拟机规范， Java 虚拟机规范规定的区域分为以下 5 个部分：

* 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成
* Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息
* 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的
* Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存
* 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据

## 196. 说一下堆栈的区别？

* 功能方面：堆是用来存放对象的，栈是用来执行程序的
* 共享性：堆是线程共享的，栈是线程私有的
* 空间大小：堆大小远远大于栈

## 197. 队列和栈是什么？有什么区别？

* 队列和栈都是被用来预存储数据的。
* 队列允许先进先出检索元素，但也有例外的情况，Deque 接口允许从两端检索元素。
* 栈和队列很相似，但它运行对元素进行后进先出进行检索

## 198. 什么是双亲委派模型？

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。

类加载器分类：

* 启动类加载器（Bootstrap ClassLoader），是虚拟机自身的一部分，用来加载Java\_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库
* 其他类加载器：
* 扩展类加载器（Extension ClassLoader）：负责加载\lib\ext目录或Java. ext. dirs系统变量指定的路径中的所有类库
* 应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类

## 199. 说一下类加载的执行过程？

类装载分为以下 5 个步骤：

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

## 200. 怎么判断对象是否可以被回收？

一般有两种方法来判断：

* 引用计数器：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题
* 可达性分析：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的

## 201. java 中都有哪些引用类型？

* 强引用：发生 gc 的时候不会被回收
* 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收
* 弱引用：有用但不是必须的对象，在下一次GC时会被回收
* 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知

## 202. 说一下 jvm 有哪些垃圾回收算法？

* 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片
* 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存
* 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半
* 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法

## 203. 说一下 jvm 有哪些垃圾回收器？

* Serial：最早的单线程串行垃圾回收器
* Serial Old：Serial 垃圾回收器的老年版本，同样也是单线程的，可以作为 CMS 垃圾回收器的备选预案
* ParNew：是 Serial 的多线程版本
* Parallel 和 ParNew 收集器类似是多线程的，但 Parallel 是吞吐量优先的收集器，可以牺牲等待时间换取系统的吞吐量
* Parallel Old 是 Parallel 老生代版本，Parallel 使用的是复制的内存回收算法，Parallel Old 使用的是标记-整理的内存回收算法
* CMS：一种以获得最短停顿时间为目标的收集器，非常适用 B/S 系统
* G1：一种兼顾吞吐量和停顿时间的 GC 实现，是 JDK 9 以后的默认 GC 选项

## 204. 详细介绍一下 CMS 垃圾回收器？

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低

## 205. 新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别？

* 新生代回收器：Serial、ParNew、Parallel Scavenge
* 老年代回收器：Serial Old、Parallel Old、CMS
* 整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收

## 206. 简述分代垃圾回收器是怎么工作的？

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

* 把 Eden + From Survivor 存活的对象放入 To Survivor 区
* 清空 Eden 和 From Survivor 分区
* From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor

　　每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。 老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程

## 207. 说一下 jvm 调优的工具？

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具

* jconsole：用于对 JVM 中的内存、线程和类等进行监控；
* jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等

## 208. 常用的 jvm 调优的参数都有哪些？

* -Xms2g：初始化推大小为 2g
* -Xmx2g：堆最大内存为 2g
* -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4
* -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2
* –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合
* -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合
* -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合
* -XX:+PrintGC：开启打印 gc 信息
* -XX:+PrintGCDetails：打印 gc 详细信息

# **20.[ReentrantLock实现原理](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=reentrantlock-%e5%ae%9e%e7%8e%b0%e5%8e%9f%e7%90%86)**

# 使用synchronized来做同步处理时，锁的获取和释放都是隐式的，实现的原理是通过编译后加上不同的机器指令来实现。

而ReentrantLock就是一个普通的类，的英文它基于AQS(AbstractQueuedSynchronizer)来实现的。

一个的英文****重入锁****：线程一个获得了锁之后仍然可以****反复****的加锁，不会出现自己阻塞自己的情况。

**AQS的英文Java并发包里实现锁，同步的一个重要的基础框架。**

## **[锁类型](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e9%94%81%e7%b1%bb%e5%9e%8b)**

ReentrantLock分为****公平锁****和****非公平锁****，可以通过构造方法来指定具体类型：

//默认非公平锁

public ReentrantLock() {

sync = new NonfairSync();

}

//公平锁

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

一般默认使用****非公平锁****，它的效率和吞吐量都比公平锁高的多（后面会分析具体原因）。

## **[获取锁](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e8%8e%b7%e5%8f%96%e9%94%81)**

通常的使用方式如下：

private ReentrantLock lock = new ReentrantLock();

public void run() {

lock.lock();

try {

//do bussiness

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

### **[公平锁获取锁](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e5%85%ac%e5%b9%b3%e9%94%81%e8%8e%b7%e5%8f%96%e9%94%81)**

首先看下获取锁的过程：

public void lock() {

sync.lock();

}

可以看到是使用sync的方法，而这个方法是一个抽象方法，具体是由其子类（FairSync）来实现的，以下是公平锁的实现：

final void lock() {

acquire(1);

}

//AbstractQueuedSynchronizer 中的 acquire()

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

第一步是尝试获取锁（tryAcquire(arg)），这个也是由其子类实现：

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

}

首先会判断AQS中的state是否等于0,0表示目前没有其他线程获得锁，当前线程就可以尝试获取锁。

****注意****：尝试之前会利用hasQueuedPredecessors()方法来判断AQS的队列中中是否有其他线程，如果有则不会尝试获取锁（****这是公平锁特有的情况****）。

如果队列中没有线程就利用CAS来将AQS中的状态修改为1，也就是获取锁，获取成功则将当前线程置置为获得锁的独占线程（setExclusiveOwnerThread(current)）。

如果state大于0时，说明锁已经被获取了，则需要判断获取锁的线程是否为当前线程（ReentrantLock支持重入），是则需要将state + 1，并将值更新。

#### **[写入队列](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e5%86%99%e5%85%a5%e9%98%9f%e5%88%97)**

如果tryAcquire(arg)电子杂志锁失败，需要则用addWaiter(Node.EXCLUSIVE)将当前线程写入队列中。

写入之前需要将当前线程包装为一个Node对象（addWaiter(Node.EXCLUSIVE)）。

**AQS中的队列是由节点节点组成的双向链表实现的。**

包装代码：

private Node addWaiter(Node mode) {

Node node = new Node(Thread.currentThread(), mode);

// Try the fast path of enq; backup to full enq on failure

Node pred = tail;

if (pred != null) {

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

首先判断队列是否为空，为不空时则将封装好的Node利用CAS写入队尾，如果并发出现失败写入就需要调用enq(node);来写入了。

private Node enq(final Node node) {

for (;;) {

Node t = tail;

if (t == null) { // Must initialize

if (compareAndSetHead(new Node()))

tail = head;

} else {

node.prev = t;

if (compareAndSetTail(t, node)) {

t.next = node;

return t;

}

}

}

}

处理这个就逻辑相当于自旋加上CAS保证一定能写入队列。

#### **[挂起等待线程](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e6%8c%82%e8%b5%b7%e7%ad%89%e5%be%85%e7%ba%bf%e7%a8%8b)**

写入队列之后需要将当前线程挂起（利用acquireQueued(addWaiter(Node.EXCLUSIVE), arg)）：

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

会首先根据node.predecessor()电子杂志到上一个节点是否为头节点，如果是则尝试获取一次锁，获取成功就万事大吉了。

如果不是头节点，或者获取锁失败，则会根据上一个节点的waitStatus状态来处理（shouldParkAfterFailedAcquire(p, node)）。

waitStatus 用于记录当前节点的状态，如节点取消，节点等待等。

shouldParkAfterFailedAcquire(p, node)返回当前线程是否需要挂起，如果需要则调用parkAndCheckInterrupt()：

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this);

return Thread.interrupted();

}

的英文他利用LockSupport的part方法来挂起当前线程的，直到被唤醒。

### **[非公平锁获取锁](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e9%9d%9e%e5%85%ac%e5%b9%b3%e9%94%81%e8%8e%b7%e5%8f%96%e9%94%81)**

公平锁与非公平锁的差异主要在获取锁：

公平锁就相当于买票，后来的人需要排到队尾依次买票，****不能插队****。

而非公平锁则没有这些规则，的英文****抢占模式****，每来一个人不会去管队列如何，直接尝试获取锁。

非公平锁：

final void lock() {

//直接尝试获取锁

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

公平锁：

final void lock() {

acquire(1);

}

还要一个重要的区别是在尝试获取锁时tryAcquire(arg)，非公平锁是不需要判断队列中是否还有其他线程，也是直接尝试获取锁：

final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

//没有 !hasQueuedPredecessors() 判断

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0) // overflow

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

## **[释放锁](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e9%87%8a%e6%94%be%e9%94%81)**

公平锁和非公平锁的释放流程都是一样的：

public void unlock() {

sync.release(1);

}

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

//唤醒被挂起的线程

unparkSuccessor(h);

return true;

}

return false;

}

//尝试释放锁

protected final boolean tryRelease(int releases) {

int c = getState() - releases;

if (Thread.currentThread() != getExclusiveOwnerThread())

throw new IllegalMonitorStateException();

boolean free = false;

if (c == 0) {

free = true;

setExclusiveOwnerThread(null);

}

setState(c);

return free;

}

首先会判断当前线程是否为获得锁的线程，由于是重入锁所以需要将state减到0才认为完全释放锁。

之后释放需要调用unparkSuccessor(h)来唤醒被挂起的线程。

## **[总结](https://crossoverjie.top/JCSprout/" \l "/thread/ReentrantLock?id=%e6%80%bb%e7%bb%93)**

由于公平锁需要关心队列的情况，得按照队列里的先后顺序来获取锁（会造成大量的线程上下文切换），而非公平锁则没有这个限制。

所以也就能解释非公平锁的效率会被公平锁更高。

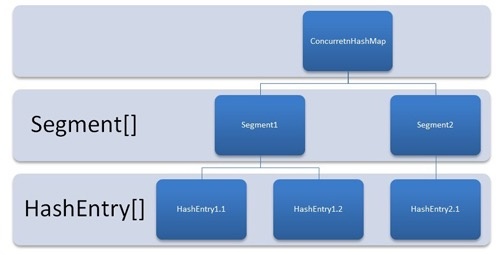
# **[ConcurrentHashMap实现原理](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=concurrenthashmap-%e5%ae%9e%e7%8e%b0%e5%8e%9f%e7%90%86)**

由于HashMap的英文一个线程不安全的容器，体现主要在容量大于总量\*负载因子发生扩容时会出现环形链表从而导致死循环。

因此需要支持线程安全的并发容器ConcurrentHashMap。

## **[JDK1.7实现](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=jdk17-%e5%ae%9e%e7%8e%b0)**

### **[数据结构](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)**



如图所示，由的英文Segment数组，HashEntry数组组成，状语从句：HashMap一样，仍然是数组加链表组成。

ConcurrentHashMap采用了分段锁技术，其中Segment继承于ReentrantLock。不会像HashTable那样不管是put还是get操作都需要做同步处理，理论上ConcurrentHashMap支持CurrencyLevel（Segment 数组数量）的线程并发。每当一个线程占用锁访问一个Segment时，不会影响到其他的Segment。

### **[得方法](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=get-%e6%96%b9%e6%b3%95)**

ConcurrentHashMap的get方法的英文非常高效的，因为整个过程都不需要加锁。

只需要将Key通过Hash之后定位到具体的Segment，再通过一次定位Hash到具体的元素上。由于HashEntry中的value属性是用volatile关键词修饰的，保证了内存可见性，所以每次获取时都是最新值（**[volatile相关知识点](https://github.com/crossoverJie/Java-Interview/blob/master/MD/Threadcore.md" \l "%E5%8F%AF%E8%A7%81%E6%80%A7" \t "_blank)**）。

### **[把方法](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=put-%e6%96%b9%e6%b3%95)**

内部HashEntry类：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

HashEntry(int hash, K key, V value, HashEntry<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

}

虽然HashEntry中的价值是用volatile关键词修饰的，但是并不能保证并发的原子性，所以把操作时仍然需要加锁处理。

首先也是通过Key的Hash定位到具体的Segment，在这里比HashMap要好的一点是：HashMap是插入元素之后再看是否需要扩容，有可能扩容之后后续就没有插入就浪费了本次扩容（扩容非常消耗性能）。

而ConcurrentHashMap不一样，它是在将数据插入之前检查是否需要扩容，之后再做插入操作。

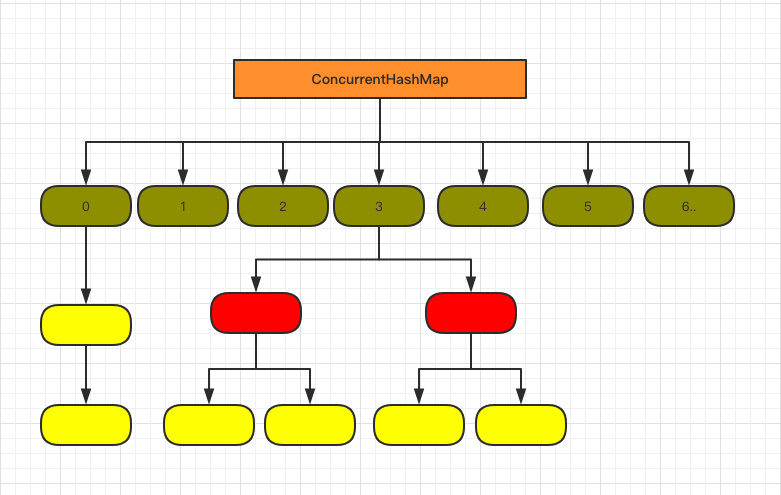
### **[大小方法](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=size-%e6%96%b9%e6%b3%95)**

每个Segment都有一个volatile修饰的全局变量count，整个是什么意思？ConcurrentHashMap的size时很明显就是将所有的count累加即可。但是volatile修饰的变量却不能保证多线程的原子性，所有直接累加很容易出现并发问题。

但每次如果调用size方法将其余的修改操作加锁效率也。很低。所以的英文做法先尝试两次将count累加，如果的容器count发生了变化再加锁来统计size。

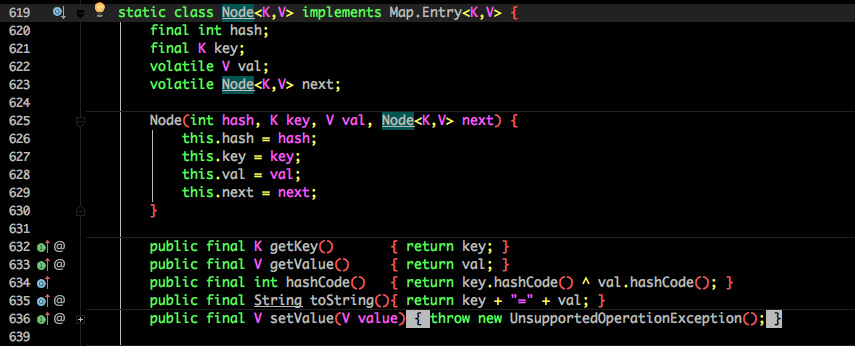
至于ConcurrentHashMap是如何知道在统计时大小发生了变化呢，每个Segment都有一个modCount变量，每当进行一次put remove等操作，modCount将会+1。只要modCount发生了变化就认为容器的大小也在发生变化。

## **[JDK1.8实现](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=jdk18-%e5%ae%9e%e7%8e%b0)**



1.8中的ConcurrentHashMap数据结构和实现与1.7还是有着明显的差异。

其中抛弃了原有的Segment分段锁，而采用了CAS + synchronized来保证并发安全性。



也将1.7中存放数据的HashEntry改为Node，但作用都是相同的。

其中的val next都用了volatile修饰，保证了可见性。

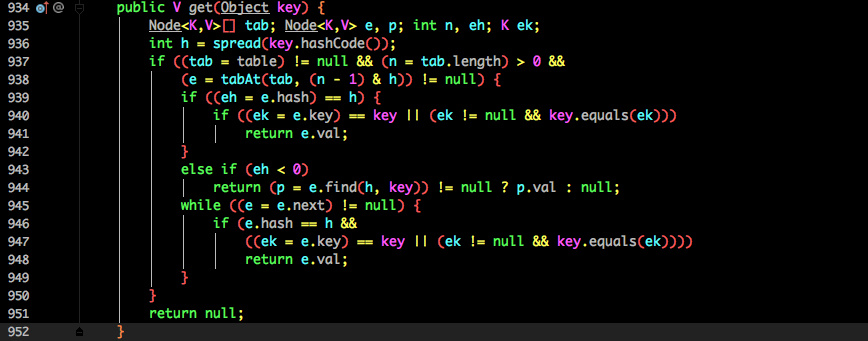
### **[把方法](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=put-%e6%96%b9%e6%b3%95-1)**

重点来看看put函数：



* 根据键计算出哈希码。
* 判断是否需要进行初始化。
* f 即为当前键定位出的节点，如果为空表示当前位置可以写入数据，利用CAS尝试写入，失败则自循保证成功。
* 如果当前位置的hashcode == MOVED == -1，则需要进行扩容。
* 如果都不满足，则利用synchronized锁写入数据。
* 如果大于数量TREEIFY\_THRESHOLD则要转换为红黑树。

### **[得方法](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=get-%e6%96%b9%e6%b3%95-1)**



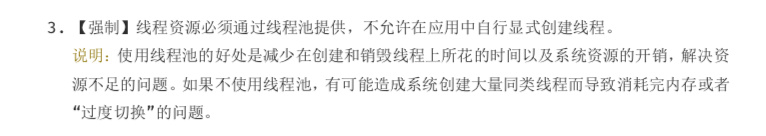
* 根据计算出来的hashcode寻址，如果就在桶上那么直接返回值。
* 如果是红黑树那就按照树的方式获取值。
* 都不满足那就按照链表的方式遍历获取值。

## **[总结](https://crossoverjie.top/JCSprout/" \l "/thread/ConcurrentHashMap?id=%e6%80%bb%e7%bb%93)**

1.8在1.7的数据结构上做了大的改动，采用红黑树之后可以保证查询效率（O(logn)），甚至取消了ReentrantLock改为了synchronized，这样可以看出在新版的JDK中对同步优化是很到位的。

# 22.如何优雅的使用和理解线程池

平时接触过多线程开发的童鞋应该都或多少少了解过线程池，之前发布的“阿里巴巴Java手册”里也有一条：



可见线程池的重要性。

简单来说使用线程池有以下几个目的：

* 线程是稀缺资源，不能频繁的创建。
* 解耦作用;线程的创建于执行完全分开，方便维护。
* 应当将其放入一个池子中，可以给其他任务进行复用。

## **[线程池原理](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e7%ba%bf%e7%a8%8b%e6%b1%a0%e5%8e%9f%e7%90%86)**

谈到线程池就会想到池化技术，其中最核心的思想就是把宝贵的资源放到一个池子中;每次使用都从里面获取，用完之后又放回池子供其他人使用，有点吃大锅饭的意思。

那在Java中又是如何实现的呢？

在JDK 1.5之后推出了相关的api，常见的创建线程池方式有以下几种：

* Executors.newCachedThreadPool()：无限线程池。
* Executors.newFixedThreadPool(nThreads)：创建固定大小的线程池。
* Executors.newSingleThreadExecutor()：创建单个线程的线程池。

其实看这三种方式创建的源码就会发现：

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>());

}

还是实际上利用ThreadPoolExecutor类实现的。

所以重点我们下来看ThreadPoolExecutor的英文怎么玩的。

首先是创建线程的api：

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)

这几个核心参数的作用：

* corePoolSize 为线程池的基本大小。
* maximumPoolSize 为线程池最大线程大小。
* keepAliveTime状语从句：unit则的英文线程空闲后的存活时间。
* workQueue 用于存放任务的阻塞队列。
* handler 当队列和最大线程池都满了之后的饱和策略。

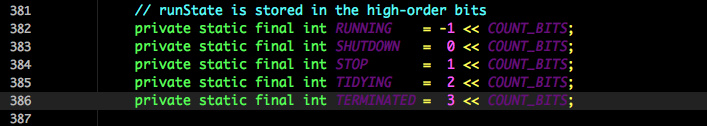
了解了这几个参数再来看看实际的运用。

通常我们都是使用：

threadPool.execute(new Job());

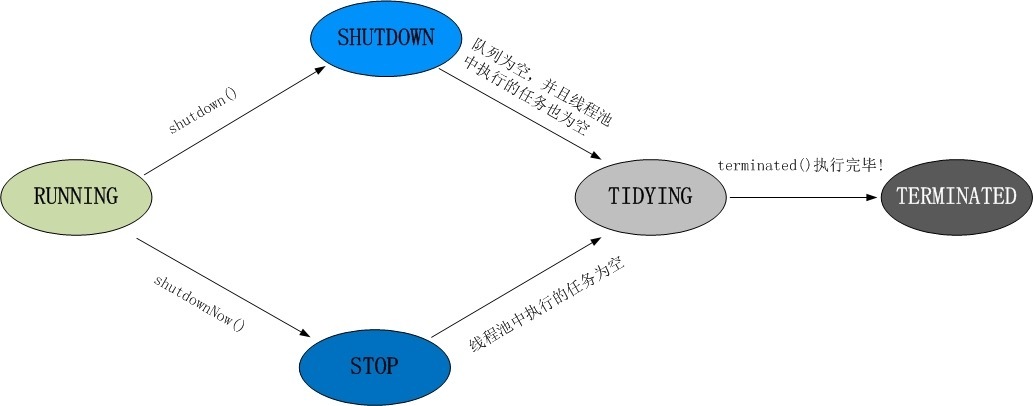
这样的方式来提交一个任务到线程池中，所以的核心就是逻辑execute()函数了。

在具体分析之前先了解下线程池中所定义的状态，这些状态都和线程的执行密切相关：

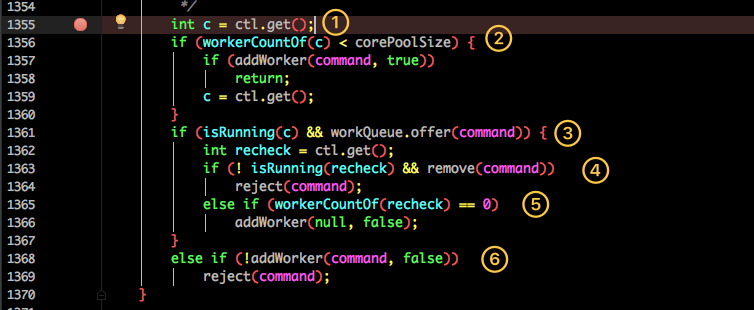


* RUNNING 自然是运行状态，指可以接受任务执行队列里的任务
* SHUTDOWN调用指了shutdown()方法，不再接受新任务了，但是队列里的任务得执行完毕。
* STOP调用指了shutdownNow()方法，不再接受新任务，同时抛弃阻塞队列里的所有任务并中断所有正在执行任务。
* TIDYING所有任务都执行完毕，调用在shutdown()/shutdownNow()中都会尝试更新为这个状态。
* TERMINATED终止状态，执行当terminated()后会更新为这个状态。

用图表示为：

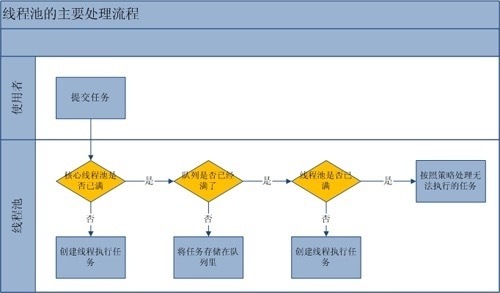


看看然后execute()方法的英文如何处理的：



1. 获取当前线程池的状态。
2. 当前线程数量小于coreSize时创建一个新的线程运行。
3. 如果当前线程处于运行状态，并且写入阻塞队列成功。
4. 双重检查，再次获取线程状态;如果线程状态变了（非运行状态）就需要从阻塞队列移除任务，并尝试判断线程是否全部执行完毕同时执行拒绝策略。
5. 如果当前线程池为空就新创建一个线程并执行。
6. 如果在第三步的判断为非运行状态，尝试新建线程，如果失败则执行拒绝策略。

这里借助“聊聊并发”的一张图来描述这个流程：



### **[如何配置线程](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e5%a6%82%e4%bd%95%e9%85%8d%e7%bd%ae%e7%ba%bf%e7%a8%8b)**

流程聊完了再来看看上文提到了几个核心参数应该如何配置呢？

有一点是肯定的，线程池肯定是不是越大越好。

通常我们是需要根据这批任务执行的性质来确定的。

* IO密集型任务：由于线程并不是一直在运行，所以可以尽可能的多配置线程，比如CPU个数\* 2
* CPU密集型任务（大量复杂的运算）应当分配较少的线程，比如CPU个数相当的大小。

当然这些都是经验值，最好的方式还是根据实际情况测试得出最佳配置。

### **[优雅的关闭线程池](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e4%bc%98%e9%9b%85%e7%9a%84%e5%85%b3%e9%97%ad%e7%ba%bf%e7%a8%8b%e6%b1%a0)**

有运行任务自然也有关闭任务，从上文提到的5个状态就能看出如何来关闭线程池。

其实无非就是两个方法shutdown()/shutdownNow()。

但他们有着重要的区别：

* shutdown() 执行后停止接受新任务，会把队列的任务执行完毕。
* shutdownNow() 也是停止接受新任务，但会中断所有的任务，将线程池状态变为停。

**两个方法都会中断线程，用户可自行判断是否需要响应中断。**

shutdownNow() 要更简单粗暴，可以根据实际场景选择不同的方法。

我通常是按照以下方式关闭线程池的：

long start = System.currentTimeMillis();

for (int i = 0; i <= 5; i++) {

pool.execute(new Job());

}

pool.shutdown();

while (!pool.awaitTermination(1, TimeUnit.SECONDS)) {

LOGGER.info("线程还在执行。。。");

}

long end = System.currentTimeMillis();

LOGGER.info("一共处理了【{}】", (end - start));

pool.awaitTermination(1, TimeUnit.SECONDS)会每隔一秒钟检查一次是否执行完毕（状态为TERMINATED），当从循环退出时就表明线程池已经完全终止了。

## **[SpringBoot使用线程池](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=springboot-%e4%bd%bf%e7%94%a8%e7%ba%bf%e7%a8%8b%e6%b1%a0)**

2018年了，SpringBoot盛行;来看看在SpringBoot中应当如何配置和使用线程池。

既然用了SpringBoot，那自然得发挥Spring的特性，所以需要Spring来帮我们管理线程池：

@Configurationpublic class TreadPoolConfig {

/\*\*

\* 消费队列线程

\* @return

\*/

@Bean(value = "consumerQueueThreadPool")

public ExecutorService buildConsumerQueueThreadPool(){

ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()

.setNameFormat("consumer-queue-thread-%d").build();

ExecutorService pool = new ThreadPoolExecutor(5, 5, 0L, TimeUnit.MILLISECONDS,

new ArrayBlockingQueue<Runnable>(5),namedThreadFactory,new ThreadPoolExecutor.AbortPolicy());

return pool ;

}}

使用时：

@Resource(name = "consumerQueueThreadPool")

private ExecutorService consumerQueueThreadPool;

@Override

public void execute() {

//消费队列

for (int i = 0; i < 5; i++) {

consumerQueueThreadPool.execute(new ConsumerQueueThread());

}

}

其实也挺简单，就是创建了一个线程池的bean，在使用时直接从Spring中取出即可。

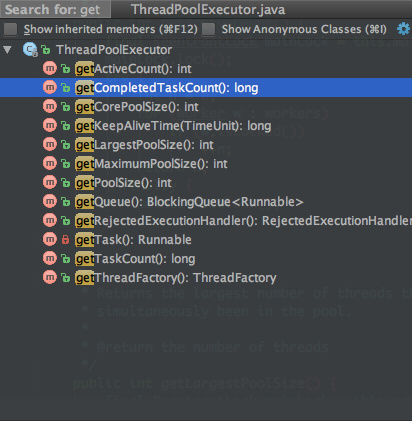
## **[监控线程池](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e7%9b%91%e6%8e%a7%e7%ba%bf%e7%a8%8b%e6%b1%a0)**

谈到了SpringBoot，也可利用它执行器组件来做线程池的监控。

线程怎么说都是稀缺资源，对线程池的监控可以知道自己任务执行的状况，效率等。

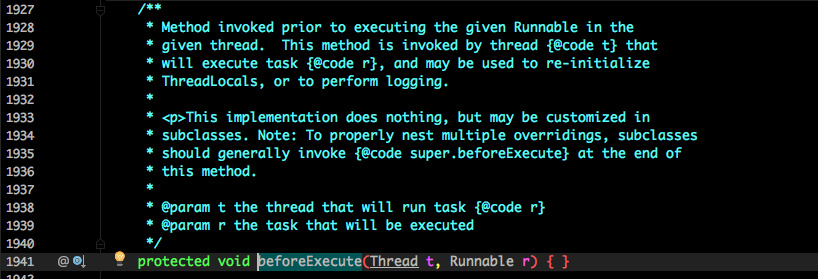
关于致动器就不再细说了，的感兴趣可以看看**[这篇](http://t.cn/ReimM0o" \t "_blank)**，有详细整理过如何暴露监控端点。

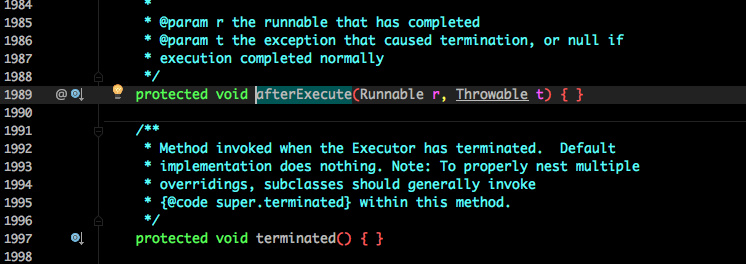
其实ThreadPool本身已经提供了不少api可以获取线程状态：



很多方法看名字就知道其含义，只需要将这些信息暴露到SpringBoot的监控端点中，我们就可以在可视化页面查看当前的线程池状态了。

甚至我们可以继承线程池扩展其中的几个函数来自定义监控逻辑：





看这些名称和定义都知道，这是让子类来实现的。

可以在线程执行前，后，终止状态执行自定义逻辑。

## **[线程池隔离](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e7%ba%bf%e7%a8%8b%e6%b1%a0%e9%9a%94%e7%a6%bb)**

**线程池看似很美好，但也会带来一些问题。**

如果我们很多业务都依赖于同一个线程池，当其中一个业务因为各种不可控的原因消耗了所有的线程，导致线程池全部占满。

这样其他的业务也就不能正常运转了，这对系统的打击是巨大的。

比如我们Tomcat接受请求的线程池，假设其中一些响应特别慢，线程资源得不到回收释放;线程池慢慢被占满，最坏的情况就是整个应用都不能提供服务。

所以需要我们将线程池****进行隔离****。

通常的做法是按照业务进行划分：

**比如下单的任务用一个线程池，获取数据的任务用另一个线程池。这样即使其中一个出现问题把线程池耗尽，那也不会影响其他的任务运行。**

### **[hystrix隔离](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=hystrix-%e9%9a%94%e7%a6%bb)**

这样的需求**[猬](https://github.com/Netflix/Hystrix" \t "_blank)**已经帮我们实现了。

**Hystrix是一款开源的容错插件，具有依赖隔离，系统容错降级等功能。**

来下面看看Hystrix简单的应用：

首先需要定义两个线程池，分别用于执行订单，处理用户。

/\*\*

\* Function:订单服务

\*

\* @author crossoverJie

\* Date: 2018/7/28 16:43

\* @since JDK 1.8

\*/public class CommandOrder extends HystrixCommand<String> {

private final static Logger LOGGER = LoggerFactory.getLogger(CommandOrder.class);

private String orderName;

public CommandOrder(String orderName) {

super(Setter.withGroupKey(

//服务分组

HystrixCommandGroupKey.Factory.asKey("OrderGroup"))

//线程分组

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("OrderPool"))

//线程池配置

.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()

.withCoreSize(10)

.withKeepAliveTimeMinutes(5)

.withMaxQueueSize(10)

.withQueueSizeRejectionThreshold(10000))

.andCommandPropertiesDefaults(

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.THREAD))

)

;

this.orderName = orderName;

}

@Override

public String run() throws Exception {

LOGGER.info("orderName=[{}]", orderName);

TimeUnit.MILLISECONDS.sleep(100);

return "OrderName=" + orderName;

}}/\*\*

\* Function:用户服务

\*

\* @author crossoverJie

\* Date: 2018/7/28 16:43

\* @since JDK 1.8

\*/public class CommandUser extends HystrixCommand<String> {

private final static Logger LOGGER = LoggerFactory.getLogger(CommandUser.class);

private String userName;

public CommandUser(String userName) {

super(Setter.withGroupKey(

//服务分组

HystrixCommandGroupKey.Factory.asKey("UserGroup"))

//线程分组

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("UserPool"))

//线程池配置

.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()

.withCoreSize(10)

.withKeepAliveTimeMinutes(5)

.withMaxQueueSize(10)

.withQueueSizeRejectionThreshold(10000))

//线程池隔离

.andCommandPropertiesDefaults(

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.THREAD))

)

;

this.userName = userName;

}

@Override

public String run() throws Exception {

LOGGER.info("userName=[{}]", userName);

TimeUnit.MILLISECONDS.sleep(100);

return "userName=" + userName;

}}

api 特别简洁易懂，具体详情请查看官方文档。

然后模拟运行：

public static void main(String[] args) throws Exception {

CommandOrder commandPhone = new CommandOrder("手机");

CommandOrder command = new CommandOrder("电视");

//阻塞方式执行

String execute = commandPhone.execute();

LOGGER.info("execute=[{}]", execute);

//异步非阻塞方式

Future<String> queue = command.queue();

String value = queue.get(200, TimeUnit.MILLISECONDS);

LOGGER.info("value=[{}]", value);

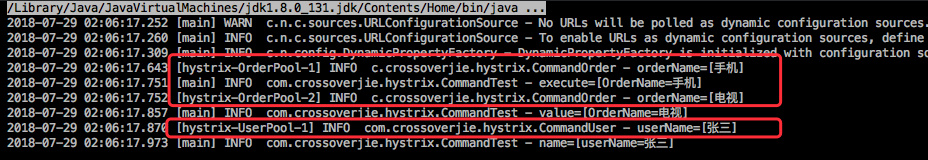
CommandUser commandUser = new CommandUser("张三");

String name = commandUser.execute();

LOGGER.info("name=[{}]", name);

}

运行结果：



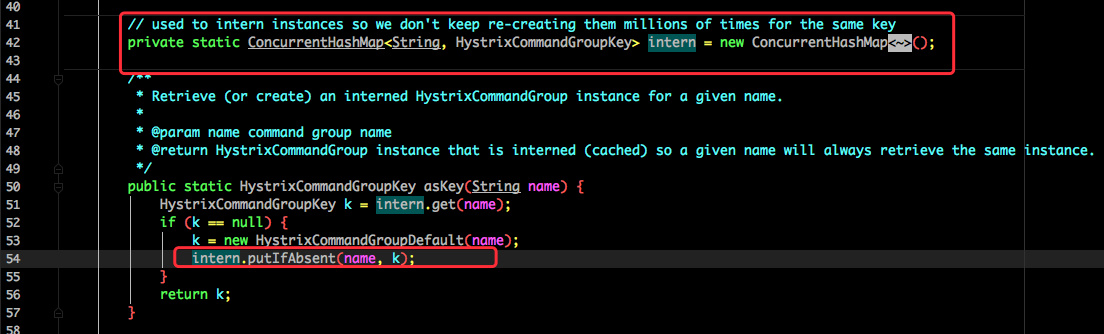
可以看到两个任务分成了两个线程池运行，他们之间互不干扰。

获取任务任务结果支持同步阻塞和异步非阻塞方式，可自行选择。

它的实现原理其实容易猜到：

**利用一个Map来存放不同业务对应的线程池。**

通过刚才的构造函数也能证明：



还要注意的一点是：

**自定义的命令并不是一个单例，每次执行需要new一个实例，不然会报This instance can only be executed once. Please instantiate a new instance.异常。**

## **[总结](https://crossoverjie.top/JCSprout/" \l "/thread/ThreadPoolExecutor?id=%e6%80%bb%e7%bb%93)**

池化技术确实在平时应用广泛，熟练掌握能提高不少效率。