# JVM篇

## Java加载类的过程

* 加载：通过一个类的全限定名称获取该类的二进制字节流>>转化为方法区的数据结构>>生成该类的class对象，作为该类的数据访问接口
* 验证：文件格式验证，字节流是否符合规范；元数据验证；字节码验证
* 准备：正式为类变量分配内存空间并设置初始值
* 解析：符号引用到直接引用的一个过程
* 初始化：真正执行定义的代码阶段
* 使用：
* 卸载：

## Java的内存分配

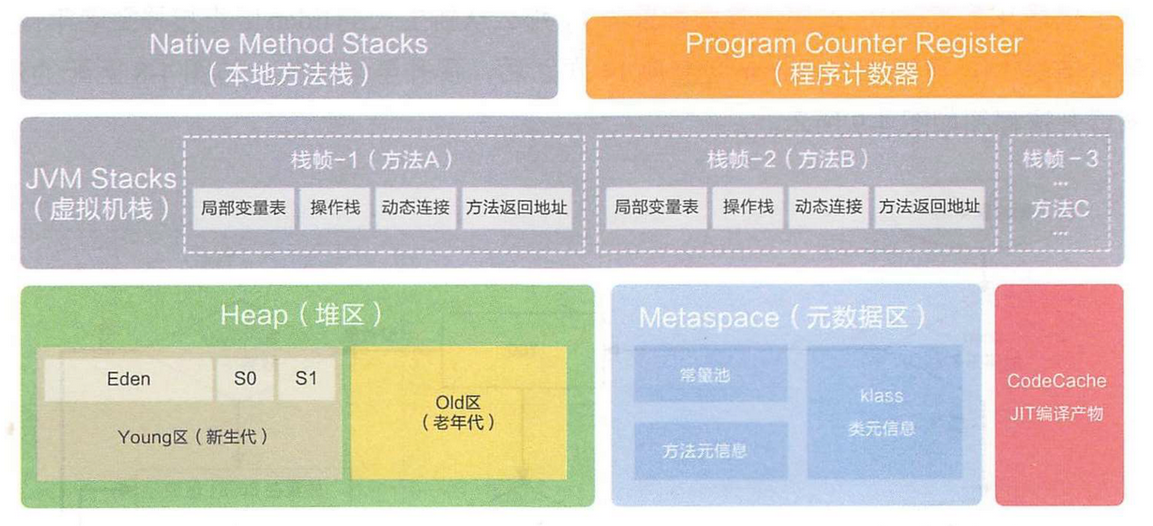
**综合网上的搜索结果，这个问题一般指Java程序运行时的内存分配，在运行时数据区内进行回答。**

**思路：**

**一个Java程序对应一个Java进程，每个Java进程包含一个或多个Java线程，每个Java进程对应唯一一个JVM实例，先回答JVM的运行时数据区的划分，哪些区域是线程共享，哪些区域是线程私有，回答每个区域的作用即可。**

回答：

（这里直接开始运行时数据区的回答），运行时数据区分为如下几个部分：



线程私有

程序计数器：内存很小的一块区域，是当前线程执行的字节码行号指示器。

虚拟机栈：虚拟机栈的内存结构有：局部变量表（预编译时期各种数据类型，对象引用等）；操作栈（也叫操作数栈，初始状态为空的桶式结构栈，可以像桶子一样嵌套，用来保存计算过程的中间结果，同时也是计算过程变量的临时存储空间）；动态链接（连接方法区中其他的类）；方法返回地址（方法运行正常结束后，将返回值传递给上层调用者，异常情况由异常处理器决定）

本地方法栈：和虚拟机栈非常相似，为native方法服务

线程共享区域

堆： 当创建一个对象时，需要在堆空间中开辟空间。堆内部空间分配如下所示：



这部分涉及了垃圾回收机制，一般来说对象先分配在E，空间满了之后触发回收 晋升到S0,S1，最后到老年代。

E 申请空间失败，则触发**Scavenge GC**

老年代/元空间/GC担保失败的时候触发 **FULL GC**

元空间(永久代/方法区/常量池)：存储类加载信息，常量，静态变量等等。

## JVM加载CLASS文件的原理机制

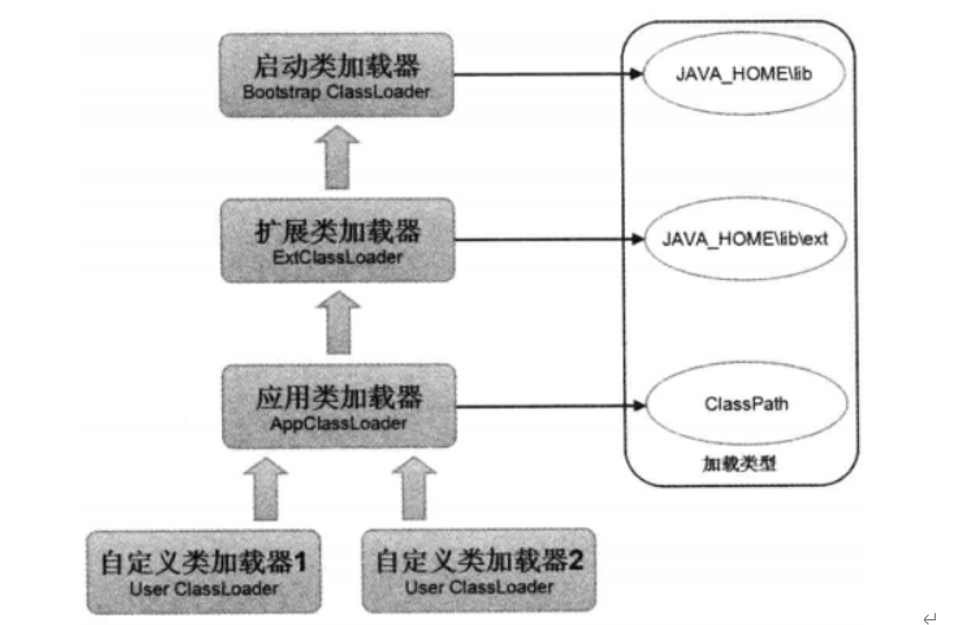
**注意：这里问的CLASS文件的加载原理机制，回答什么是CLASS文件，这个文件的加载过程（即类的加载过程），回答和类文件加载相关的类加载器（CLASS LOADER）**

**回答：**

**代码写完之后会编译为.class文件，JVM加载这个类文件的时候执行以下几个步骤：**

* **加载**
* **验证**
* **准备**
* **解析**
* **初始化**
* **使用**
* **卸载**

**执行这些步骤的工具就是 类加载器,继承关系和作用如下：**



**在Java中 类的加载器和类本身一同确认其在JVM中的唯一性。**

**加载类有两种方式：**

* **隐式加载：new 创建新对象**
* **显示加载：通过反射class.forName方法把类加载到JVM中**

这里发散记忆：反射的相关知识

## **GC是什么？**

**提示：这里需要回答GC的字面意思，GC使用的算法，机制，可以适当展开垃圾回收器**

**回答：**

GC，垃圾回收。主要回收堆区域的对象，堆分为Eden，S0,S1和老年代，在E申请空间失败的时候触发scavenge gc,存活下来的对象晋升到下一个区域，经历了在经历15次后会放入老年代（大对象直接进入老年代）。老年代空间满了之后/GC担保失败/元空间满了之后触发FULL GC

判断一个对象是否能回收常用的两种方式：引用计数法（无法解决循环依赖）和可达性分析（Java使用的，从GC ROOTS开始寻找，当对象没有和任何GC ROOTS引用链关联时就会被回收）

GC回收有对应的实现方式（算法）：

* 标记-清除：对于可回收的对象，标记结束后予以回收（这种方式会产生内存碎
* 复制算法：空间一分为二，一次只使用一片。GC结束后存活的对象移动到另外一块内存上（内存使用空间缩小）
* 标记-整理：对象回收后，将存活下来的对象移动到连续的一片内存区域上（效率不高，整理所有存活对象的引用地址）
* 分代收集，根据对象生命周期，将堆分为新生代和老年代。根据特点，新生代采用复制算法（存活下来的对象少），老年代对象存活率高，使用标记整理。

## Java中的内存泄漏

**提示：回忆Java的运行时数据区类型，内存泄漏的概念（内存没有得到正确的回收，最后不断申请新的空间，最后导致内存溢出，即内存超过了设定范围），泄漏是因为没有正确回收，所以考虑Java的回收机制进行回答**

Java的回收机制保证了当对象没有关联任何一个GC ROOTS时，就会被回收，内存得到释放。所以内存泄漏的一种情况就是一个拥有长生命周期的对象A持有一个短生命周期的对象b，b使用过一次后就再也没有使用了，同类型的对象特别多的时候，就可能产生内存泄漏。比如系统本地缓存，缓存了大量过期内容。

检查方法：一定要让程序将各个分支情况都执行完整到程序结束，在来判定对象是否被使用过，才能判断是否泄漏

HashSet会出现内存泄漏，因为是根据哈希code去进行寻址，如果修改了参与哈希code计算的值，那么调用删除的时候就无法删除了。

## 什么是分布式垃圾回收

概念：

1)Java虚拟机中，一个远程对象不仅会被本地虚拟机内的变量引用，还会被远程引用。

2)只有当一个远程对象不受到任何本地引用和远程引用，这个远程对象才会结束生命周期。

说明：

1)服务端的一个远程对象在3个地方被引用：

1>服务端的一个本地对象持有它的本地引用

2>服务端的远程对象已经注册到rmiregistry注册表中，也就是说，rmiregistry注册表持有它的远程引用。

3>客户端获得远程对象的存根对象，也就是说，客户端持有它的远程引用。

2)服务端判断客户端是否持有远程对象引用的方法：

1>当客户端获得一个服务端的远程对象的存根时，就会向服务器发送一条租约(lease)通知，以告诉服务器自己持有了这个远程对象的引用了。

2>客户端定期地向服务器发送租约通知，以保证服务器始终都知道客户端一直持有着远程对象的引用。

3>租约是有期限的，如果租约到期了，服务器则认为客户端已经不再持有远程对象的引用了。

RMI使用DGC（分布式垃圾回收）来做自动垃圾回收。DGC使用了引用计数法给远程对象提供自动内存管理

## 什么是Java内存模型

**提示：Java内存模型定义了程序中各个变量的访问规则，尝试从内存，工作内存的数据存取流程进行回答**

## Volatile有什么特点？是否是线程安全？

**提示：加了这个关键字后，内存指令有什么变化？线程是否安全？答案是否定的，多线程同时进行计算**

# Java基础篇

## Finalize()方法什么时候被调用？析构函数（finalization）的目的是什么？

java提供finalize()方法,当垃圾回收器准备释放内存的时候,会先调用finalize()方法

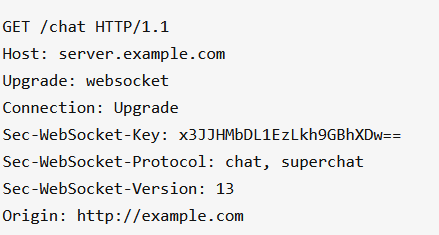
垃圾回收器（garbage collector）决定回收某对象时，就会运行该对象的finalize()方法；

finalize()方法也叫收尾方法。只会被调用一次，在调用这个方法的时候可以复活某些对象。

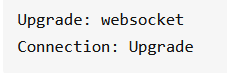
另外 JNI调用no-Java程序的时候，这个方法可以用来回收这部分内存。

## 什么是WebSocket

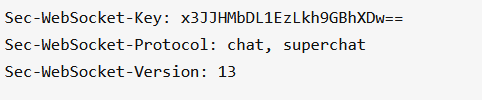
Websocket是一个位于应用层的协议，借用了http协议来完成一部分握手。



熟悉HTTP的童鞋可能发现了，这段类似HTTP协议的握手请求中，多了几个东西。



这个是告诉服务器此次协议是Websocket



第一个Base64 encode的值，随机生成，验证服务器是否是websocket

第二个用户自定义字符串，区分不同的URL

第三个是协议版本

全双工-客户端和服务器通信相互独立

# 并发编程篇

## Synchronize用过吗？简述一下

**提示：简述这个关键字的作用，使用方法，修饰代码块、方法开始回答；然后回答锁的底层，可重入性，偏向锁，轻量级锁，重量级锁，膨胀过程**

## ****为什么synchronize锁是非公平锁？****

**提示：从获取锁的流程上进行回答**

## ****什么是锁消除和锁粗化****

**提示：从JVM对代码的优化上进行回答。消除的意思是分析代码后，发现没有竞争，则取消了代码中的锁操作；粗化是指扩大了锁的范围，如果一系列连续的操作都对同一个对象反复加锁和解锁，那么需要适当的扩大锁的范围**

## ****为什么synchronize是一个悲观锁？乐观锁又是什么****

**提示：从实现上进行回答：互斥。无论是否产生竞争都会进行加锁。乐观锁的核心思想是认为数据没有竞争，可以先进行操作，操作结束后进行判断数据是否冲突，冲突了进行补偿操作，核心算法是CAS（内存值，预期值，新值），内存值和预期值相等才能进行替换**

## ****ReentrantLock和synchronize的区别****

**提示：从底层原理回答**

## ****AQS是什么？****

**提示：先简述AQS的作用（构建锁和同步工具的框架），核心思想，再简单说明哪些锁使用了这个框架，再详细介绍底层实现原理，关键点 voliate 修饰的int类型 state**

## ****ReentrantLock如何实现可重入****

**提示：从底层实现来看，使用的AQS框架，AQS框架如何实现的可重入 当前线程持有锁，state+1**

## ****请谈谈ReadWriteLock和StampedLock****

**提示：从API特性入手，先说各自的特点，再说实现区别**

## ****如何让线程彼此同步****

**提示：线程同步，又叫线程通信/协作。考虑以下几种情况:**

* **当前线程等子线程全部执行完才能继续往下走（countdownlatch，Semphore，steam流式API的多线程版本后跟.join方法）**
* **当前线程执行完，唤醒其他线程（Object的wait和notify，ReentrantLock的Condition的signal和await方法）**

## Java中的线程池如何实现的

**提示：先说线程池创建的方法ThreadPoolExecutor的参数：**

* **核心线程数**
* **最大线程数**
* **线程空闲时间**
* **时间单位**
* **等待队列**
* **创建线程的工程**
* **拒绝策略**

# Spring

## 什么是JavaConfig

**提示：从JavaConfig的来历，使用方法，规则进行回答**

**回答：**

Spring早期，是使用XML文件来定义bean和bean之间的规则。在Spring3之后，Spring提出了JavaConfig这种类型安全的Bean装配方式，使得装配方式更为灵活。

规则

规则一：@Configuration注解

　　我们在定义JavaConfig类时，都会在其上加注@Configuration注解，来表明这是一个配置类，@Configuration注解底层是@Component注解，而且这个注解会被AnnotationConfigApplicationContext来进行加载，AnnotationConfigApplicationContext是ApplicationContext的一个具体实现，代表依据配置注解启动应用上下文。

规则二：@ComponentScan注解

　　我们使用JavaConfig的目的是为了实现以前XML配置实现的功能，首先就是组件扫描功能，将我们使用特定注解标注的类统一扫描加载到Spring容器，这一功能就是依靠@ComponentScan注解来实现的，我们可以为其指定位置参数来指定要扫描的包。

规则三：@Bean注解

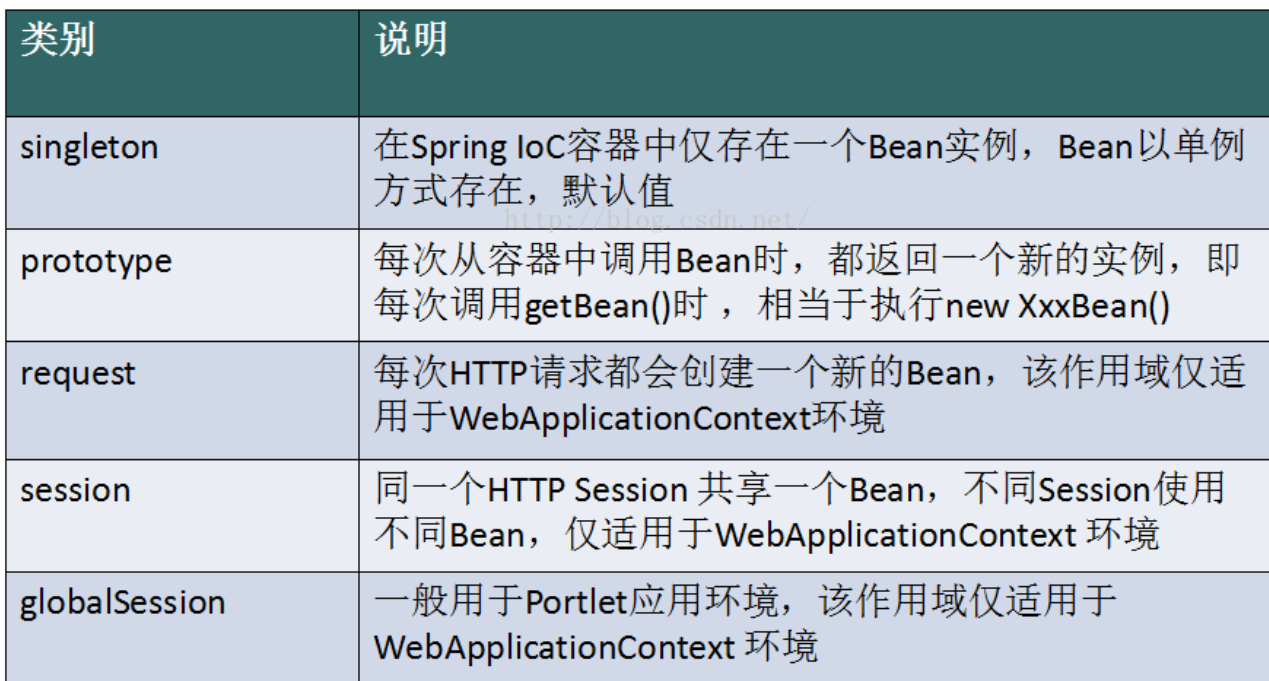
　　使用@Bean注解我们可以实现XML配置中手动配置第三方Bean的功能，这里我们使用方法来定义Bean，并在方法前面加注@Bean注解，表示要将该方法返回的对象加载到Spring容器中，这样就对我们的方法定义带来了一些限制，这些限制包括方法的大概格式：

　　　　1-方法带返回值，且返回类型为你要加载的第三方类类型

　　　　2-方法的名称为默认的Bean的name，如果要自定义Bean的name，可以使用@Bean注解的name属性。

　　　　3-要实现注入只需要将要注入的Bean的类型作为参数，调用该类的带参数的构造器构建这个Bean，或者采用第二种方式：先创建这个类的对象，然后调用该对象的set方法进行注入，以被注入的Bean的方法为参数

## Spring bean的作用域



## 简述一下spring bean的生命周期

**提示：这里应该联想到与bean相关的接口，各种aware接口， beanPostProcessor接口，beanactory**



实例化前 接口 InstantiationAwareBeanPostProcessor 的

postProcessBeforeInstantiation

实例化后阶段 postProcessAfterInitialization

populateBean阶段 注入各种依赖

aware接口回调

初始化阶段前 BeanPostProcessor接口postProcessBeforeInitialization

后 postProcessAfterInitialization

## Beanfactory和applicationConxt的区别

## 自动装配模式的区别

**提示：回答几种依赖注入的实现方式**

## 常用的bean自动注入的几个注解的区别？

## Spring框架提供了那些不同的事件类型

## Spring事务传播

Spring定义了七种传播行为：



PROPAGATION\_REQUIRED：

如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

可以把事务想像成一个胶囊，在这个场景下方法B用的是方法A产生的胶囊（事务）。

PROPAGATION\_SUPPORTS

如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。

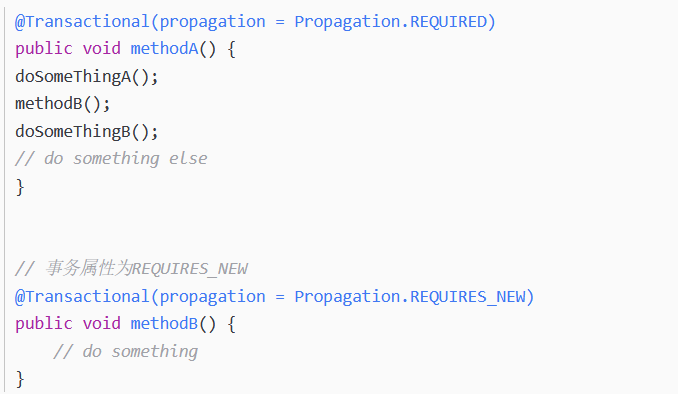
PROPAGATION\_MANDATORY

如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

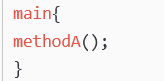
PROPAGATION\_REQUIRES\_NEW

使用PROPAGATION\_REQUIRES\_NEW,需要使用 JtaTransactionManager作为事务管理器。

它会开启一个新的事务。如果一个事务已经存在，则先将这个存在的事务挂起。



当调用



相当于



把ts1称为外层事务，ts2称为内层事务。从上面的代码可以看出，ts2与ts1是两个独立的事务，互不相干。Ts2是否成功并不依赖于 ts1。如果methodA方法在调用methodB方法后的doSomeThingB方法失败了，而methodB方法所做的结果依然被提交。而除了 methodB之外的其它代码导致的结果却被回滚了

PROPAGATION\_NOT\_SUPPORTED

总是非事务地执行，并挂起任何存在的事务。使用PROPAGATION\_NOT\_SUPPORTED,也需要使用JtaTransactionManager作为事务管理器。

PROPAGATION\_NEVER

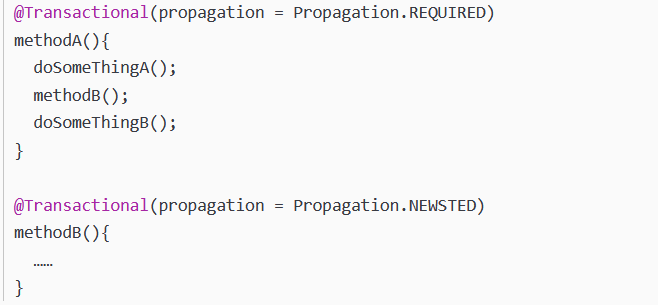
总是非事务地执行，如果存在一个活动事务，则抛出异常。

PROPAGATION\_NESTED

如果一个活动的事务存在，则运行在一个嵌套的事务中。 如果没有活动事务, 则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行。

这是一个嵌套事务,使用JDBC 3.0驱动时,仅仅支持DataSourceTransactionManager作为事务管理器。

需要JDBC 驱动的java.sql.Savepoint类。使用PROPAGATION\_NESTED，还需要把PlatformTransactionManager的nestedTransactionAllowed属性设为true(属性值默认为false)。



如果单独调用methodB方法，则按REQUIRED属性执行。如果调用methodA方法，相当于下面的效果：



当methodB方法调用之前，调用setSavepoint方法，保存当前的状态到savepoint。如果methodB方法调用失败，则恢复到之前保存的状态。但是需要注意的是，这时的事务并没有进行提交，如果后续的代码(doSomeThingB()方法)调用失败，则回滚包括methodB方法的所有操作。嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

# Spring MVC

## Spring MVC的工作原理

**提示：简述请求流程**

## ****MVC常用注解解答****

# Spring boot

## 如何重新加载Spring boot上的更改，而无需重新启动服务器

依赖于spring boot devtool

## Spring Boot监视器是什么？

监视器指的就是spring boot actuator，是spring启动框架重要的功能之一。监控器公开了一组可直接作为http URL访问的reset端点来检查状态，能够健康检查，审计，统计和http追踪。

## Spring batch是什么

Spring Batch是用来处理大量数据操作的一个框架，主要用来读取大量数据，然后进行一定处理后输出成指定的形式。

Spring Batch主要由以下几部分组成：

* JobRepository：用来注册Job的容器
* JobLauncher：用来启动Job的接口
* Job：我们要实际执行的任务，包含一个或式个Step
* Step：Step-步骤包含ItemReader、ItemProcessor和ItemWriter
* ItemReader：用来读取数据的接口
* ItemProdessor：用来处理数据的接口
* ItemWriter：用来输出数据的接口

以上SpringBatch的主要组成部分只需注册成Spring的Bean即可。若想开启批处理的支持还需在配置类上使用@EnableBatchProcessing。

需要在数据库中初始化与spring batch相关的表。

## 如何优雅的处理异常

**提示：这里涉及到了spring的全局异常处理和异常处理相关注解**

**方法一：**

**@ControllerAdvice：异常集中处理，更好的使业务逻辑与异常处理剥离开，可以指定某一个类为异常处理类，并包含各种类型的异常处理，作用在类上。**

**@ExceptionHandler：统一处理某一类异常，从而能够减少代码重复率和复杂度，并生成对应的映射关系，作用在方法上。**

**方法二：**

**还可以使用AOP结合注解@AfterThrowing进行异常拦截**

# 微服务

# MySQL

# Redis

## 什么是缓存穿透

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

## 什么是缓存雪崩

缓存一定时间后集体失效，所有缓存全部打到了数据库上

**解决方法：**

加锁排队：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待；

数据预热：可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀；

做二级缓存，或者双缓存策略：Cache1 为原始缓存，Cache2 为拷贝缓存，Cache1 失效时，可以访问 Cache2，Cache1 缓存失效时间设置为短期，Cache2 设置为长期。

在缓存的时候给过期时间加上一个随机值，这样就会大幅度的减少缓存在同一时间过期。

## ****怎么保证缓存和数据库数据一致性****

先更新数据库，后更新缓存

先更新缓存，后更新数据库

先删除缓存，后更新数据库

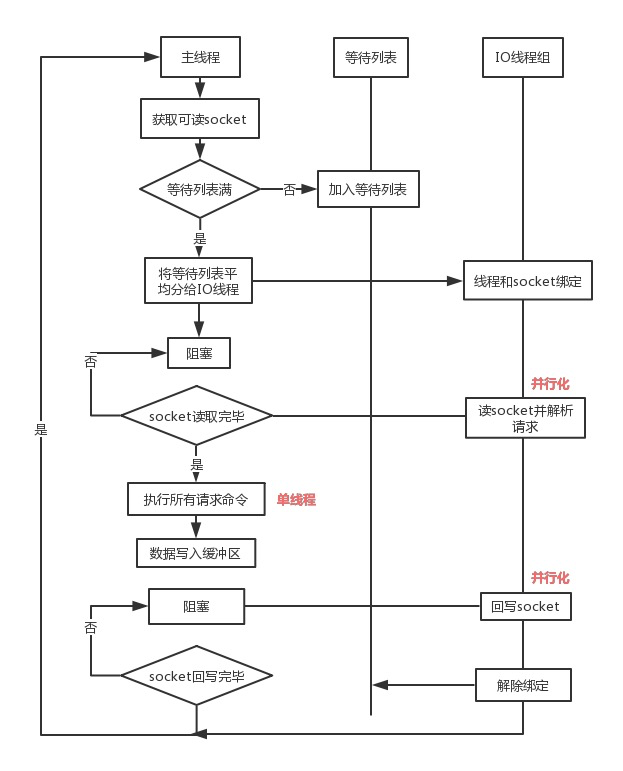
先更新数据库，后删除缓存

## 优化方式

**提示：Redis就是管理key-value的工具，优化就是要针对这个进行优化。方向一：控制key的数量；方向二：缩减key和value的长度；**

## ****Redis 6.0 多线程机制？****

提示：类似于netty服务端，设置两个线程池



# ES

# Tomcat性能优化