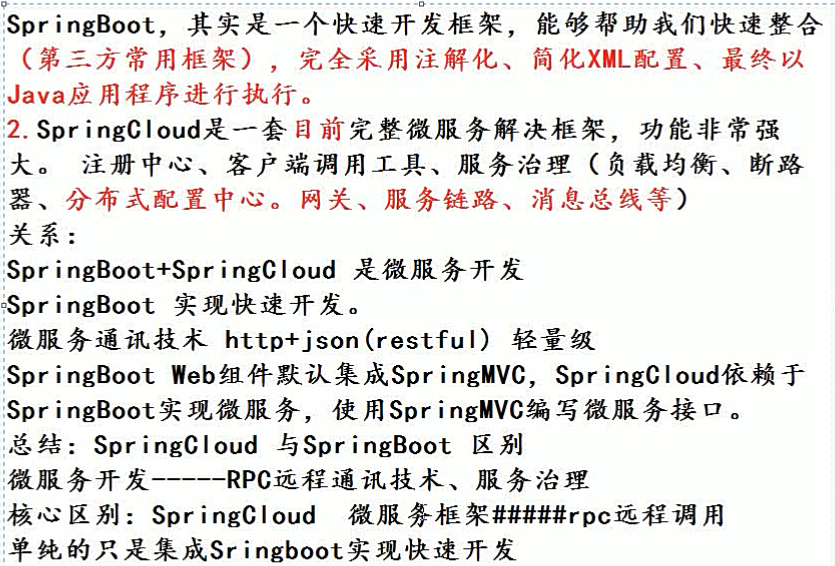
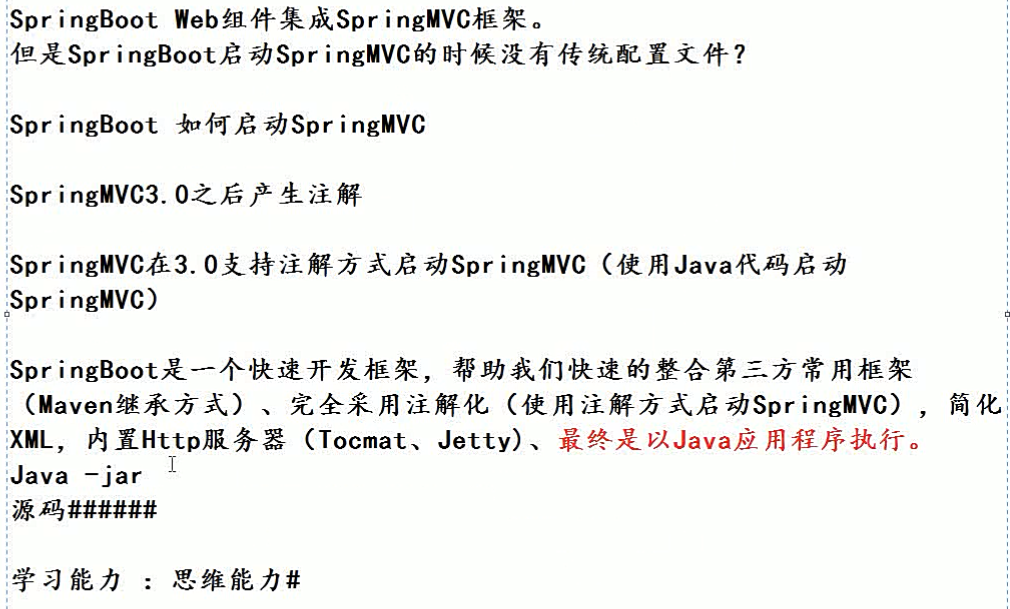


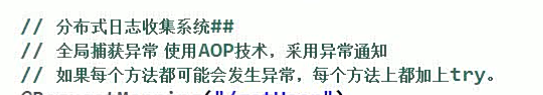
spring-boot-starter-parent依赖主要是用来控制引入依赖包的版本控制



5.



1. **SpringCloud 依赖Springboot实现微服务，Springboot默认继承SpringMVC, SpringCloud使用SpringMVC编写http接口，微服务通讯接口（http+json）**
2. **Springboot对内置的tomcat不支持jsp，需要外部引入tomcat**
3. **日志信息收集**



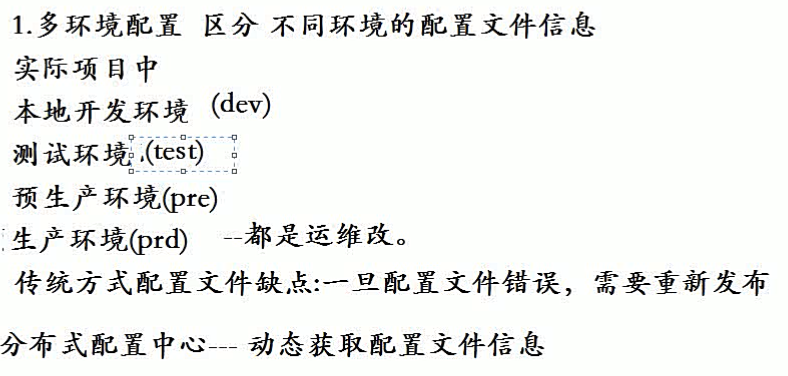
1. **springboot集成lombok,底层使用字节码技术 ASM，修改字节码文件，生成get和set**
2. **使用@EnableAsync和@Async实现异步调用方法：使用AOP在运行时创建单独线程调用方法**
3. **创建线程的2种方式，一种是直接继承Thread，另外一种就是实现Runnable接口。   
   这2种方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。   
   如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，这样使用起来就比较麻烦。从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。**
4. **多环境配置，一般环境分为：**

**本地开发环境：dev**

**测试环境：test**

**预发布环境：pre**

**生产环境：prd**





1. **Springboot整个事务管理**

**事务分类：**

**声明式事务；**

**编程式事务**

**声明式事务原理：使用aop技术的环绕同时进行拦截**

**为什么不用try-catch：**

**抛出异常，使得事务遇到异常时可以回滚**

**Springboot默认整合事务**

1. **Springboot整个多数据源**

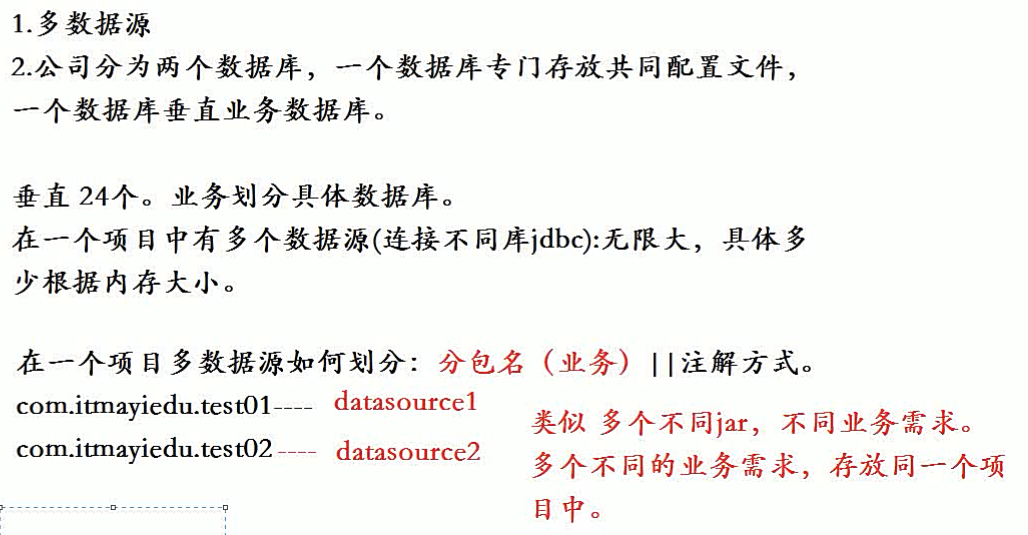
**多数据源的需求：在一个项目中可能因为不用的业务需求使用不同的数据库，比如有些项目的共用配置文件放在一个数据库，另一个为垂直业务数据库。**

**数据源的划分：按照包名划分（因为业务划分同时是通过分包实现）**

**按照注解划分**

**com.smile.springboot.item1 🡪datasource1**

**com.smile.springboot.item2 🡪datasource2**

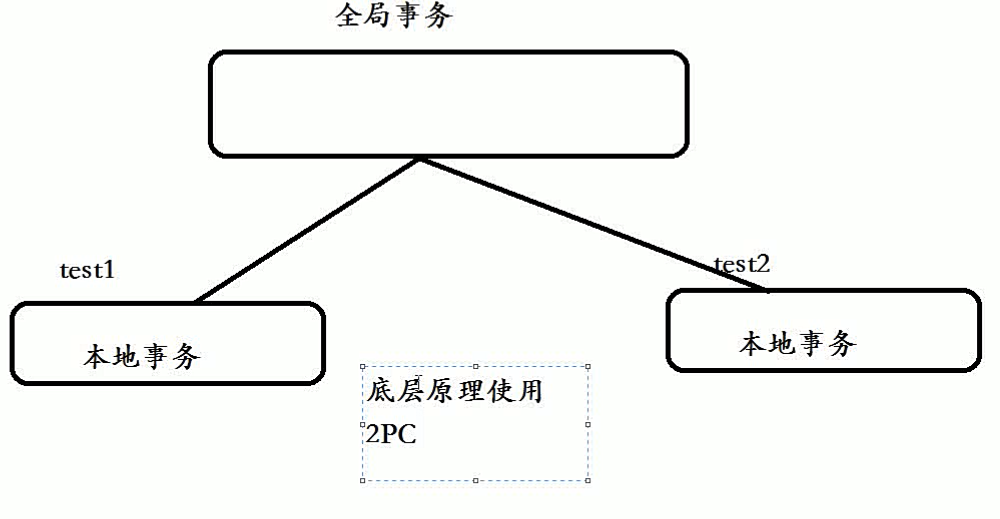


**多数据源的业务场景：**

项目底层有正常业务库和日志库，希望解决的是将项目中的一些日志单独记录到一个库里，比如用户操作记录、产品更新记录等。

说一下为什么会有这个需求：用户操作记录和产品更新记录可能很多，而实际中使用的又很少，就只是在某些页面单独展示一下操作或更新记录，绝大部分时间都在不停的做着插入操作，这时就可以把这种记录放到业务核心库外面。

1. **jta+atomikos实现分布事务解决方案**



### 本地事务和全局事务

如果查看Spring事务相关的文档，经常会看到’local transactions’和’external transactions’，本地事务和全局事务(或叫外部事务)。上面我们说了，对任何资源，只要它提供了事务的操作，我们就能使用spring的事务管理来提供事务。由于spring提供了一个事务管理的抽象接口，而事务的控制，可以是在spring容器来控制，也可以由外部的事务管理模块来控制，这就是本地事务和全局事务的区别。

本地事务就是指的是由Spring容器创建和维护的事务。例如在使用JDBC事务操作数据库的时候，spring容器会在需要的时候创建事务的上下文，开启一个JDBC的事务，然后调用业务方法，执行完成后，调用commit方法；然后在出错的时候调用资源的rollback方法。还有事务的传播、隔离等也都是由Spring容器来提供。本地事务只能针对一个资源实现完全的事务控制。如果要在一个本地事务中操作两个资源（例如两个数据库），实际上先后在两个数据库的Connection上调用commit()方法去提交。

而外部事务，就是spring只负责通过事务的接口来开始事务、提交事务、回滚事务，而具体的操作还是得有外部提供的事务管理的模块或组件来执行和维护。例如我们使用JBoss来运行我们的web应用，然后在JBoss上配置了JTA的事务。那么事务的具体管理和维护就是由JBoss提供的事务管理模块来进行。

本地事务和外部事务的一个主要区别就是，是否能对多个资源实现事务控制。

1. **XA协议**

### XA

XA是由X/Open组织提出的分布式事务的架构（或者叫协议）。XA架构主要定义了（全局）事务管理器（Transaction Manager）和（局部）资源管理器（Resource Manager）之间的接口。XA接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。也就是说，在基于XA的一个事务中，我们可以针对多个资源进行事务管理，例如一个系统访问多个数据库，或即访问数据库、又访问像消息中间件这样的资源。这样我们就能够实现在多个数据库和消息中间件直接实现全部提交、或全部取消的事务。XA规范不是java的规范，而是一种通用的规范，

目前各种数据库、以及很多消息中间件都支持XA规范。  
JTA是满足XA规范的、用于Java开发的规范。所以，当我们说，使用JTA实现分布式事务的时候，其实就是说，使用JTA规范，实现系统内多个数据库、消息中间件等资源的事务。

### 什么是JTA

JTA(Java Transaction API)，是J2EE的编程接口规范，它是XA协议的JAVA实现。它主要定义了：

* 一个事务管理器的接口javax.transaction.TransactionManager，定义了有关事务的开始、提交、撤回等操作。
* 一个满足XA规范的资源定义接口javax.transaction.xa.XAResource，一种资源如果要支持JTA事务，就需要让它的资源实现该XAResource接口，并实现该接口定义的两阶段提交相关的接口。

如果我们有一个应用，它使用JTA接口实现事务，应用在运行的时候，就需要一个实现JTA的容器，一般情况下，这是一个J2EE容器，像JBoss，Websphere等应用服务器。但是，也有一些独立的框架实现了JTA，例如[Atomikos](https://www.atomikos.com/), [bitronix](https://github.com/bitronix/btm)都提供了jar包方式的JTA实现框架。这样我们就能够在Tomcat或者Jetty之类的服务器上运行使用JTA实现事务的应用系统。

在上面的本地事务和外部事务的区别中说到，JTA事务是外部事务，可以用来实现对多个资源的事务性。它正是通过每个资源实现的XAResource来进行两阶段提交的控制。感兴趣的同学可以看看这个接口的方法，除了commit, rollback等方法以外，还有end(), forget(), isSameRM(), prepare()等等。光从这些接口就能够想象JTA在实现两阶段事务的复杂性。

### 幂等性原则

上面说了分布式事务的原则，以最终一致性为目标，那为了实现这个目标，我就需要很多异常情况的处理，包括数据库失败、业务代码失败、网络错误等。举例来说，在一个接口调用，如果发生超时，我就需要重试。但是也有可能对方的服务已经处理完这个请求，只是在完成返回结果的时候，网络传输的问题导致超时。那么服务调用端再重试，实际上就是发了两次请求。所以，我就需要对于分布式服务的事务处理，对于同样的消息，只会处理一次。

分布式系统的幂等性，就是对于一个处理接口，如果它会对系统造成副作用，也就是修改数据，那就需要保证对于同样的请求，不管请求了多少次，结果都是一致的。

那么，如何保证这个幂等性呢？一种比较通用的方法就是，对每一个请求，生成一个token，而且需要唯一，然后将这个token放在请求的参数里面。服务在处理这个请求之前，先拿到token，检查这个token是否已经处理过，只有没有处理过的才去处理。这个token可以保存在数据库、redis、甚至内存等地方。由于它只是用来记录已经处理的请求的token，所以大可不必保存在内存中。由于在分布式系统中，一个服务会部署多个，一个请求失败后重新发送，有可能是被发送到另一台机器上。所以这个token应该是服务范围共享的，我们需要在同一个服务的多个部署都能共享访问的地方，来保存已经处理过的token。

所以，使用redis来保存token是一个不错的选择。

1. **分页插件**



1. **springboot热部署配置**

**原理：使用类加载器（classloader重新读取字节码文件到jvm）**

spring-boot-devtools 是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要更快，更快指的不是节省出来的手工操作的时间。

其深层原理是使用了两个ClassLoader，一个Classloader加载那些不会改变的类（第三方Jar包），另一个ClassLoader加载会更改的类，称为 restart ClassLoader

,这样在有代码更改的时候，原来的restart ClassLoader 被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间（5秒以内）

**实现：**

1. **监听class文件是否改变（根据版本号或修改时间）**
2. **如果class文件发生改变，classloader重新读取**

**热部署不适合应用与生产环境——不安全**

**主要应用与本地的开发测试，提高开发效率**

1. **Springboot性能调优**
2. **优化Springboot启动速度：扫包优化**
3. **JVM参数调优：优化整体吞吐量**

**优化策略：初始堆内存和最大堆相同**



**注意外部启动和内部启动的区别**

**内部启动：直接在idea中启动**

**外部启动：将项目打成jar包，使用java命令启动**

**在idea将Springboot项目打成jar包：**<https://www.cnblogs.com/xugh/p/9680256.html>

**使用jconsole查看Springboot项目运行的堆内存大小**

**在运行jar包的时候，启动jdk/bin/jconsole.exe查看**

**使用Apache的jmeter可以进行压力测试：**

**出去内置tomcat:**

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 <exclusions>  
 <exclusion>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-tomcat</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>

**引入undertow:**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-undertow</artifactId>  
</dependency>

**分别使用tomcat和undertow测试10000次请求，查看吞吐量，测试报告：**

|  |  |  |  |
| --- | --- | --- | --- |
| **服务器** | **第一次/s** | **第二次** | **第三次** |
| **tomcat** | **582.3** | **832.6** | **462** |
| **undertow** | **445** |  |  |

1. **yml格式的配置文件：简约，结构性强**

**冒号后面带一个空格；**

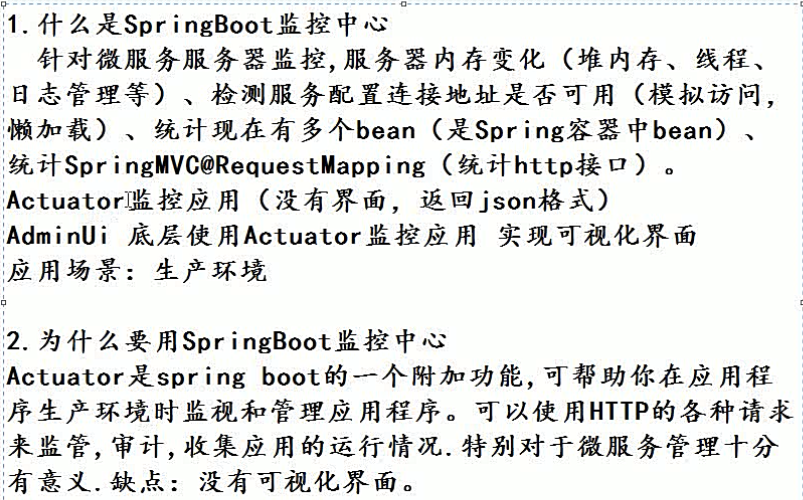
**次级在前面加两个空格**

1. **使用mvn clean package命令将Springboot打成jar或war（能够访问jsp）包，放到外部tomcat的安装目录的webapp目录下，实现外部启动**

**存在问题：**

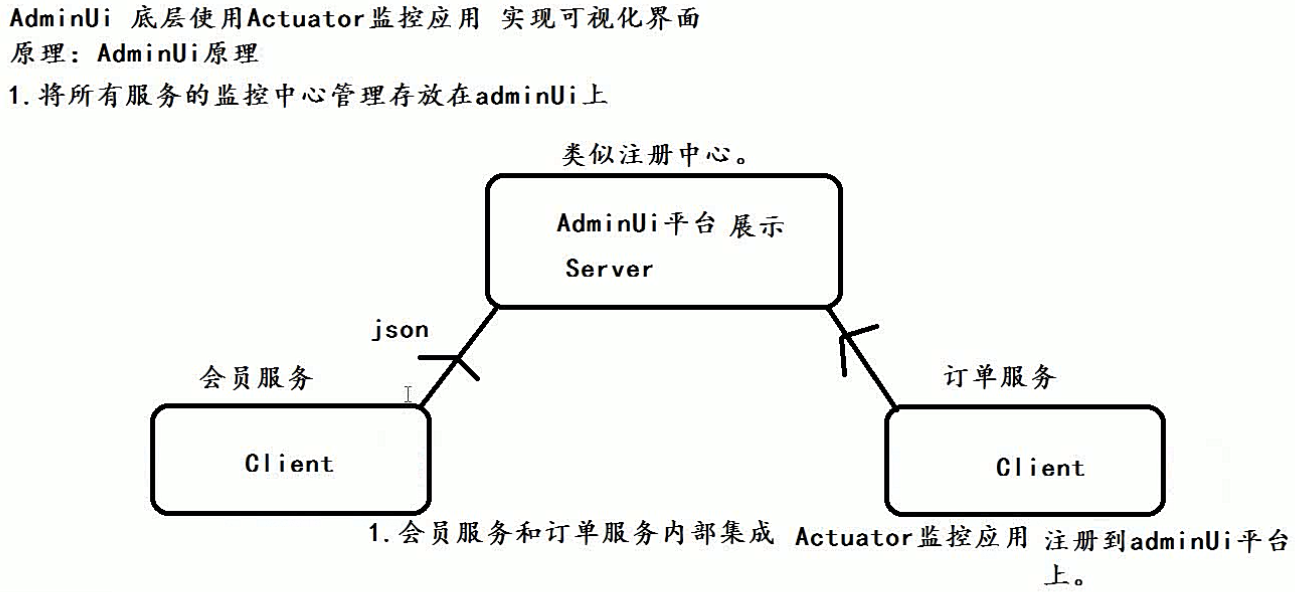


1. **actuator概述**

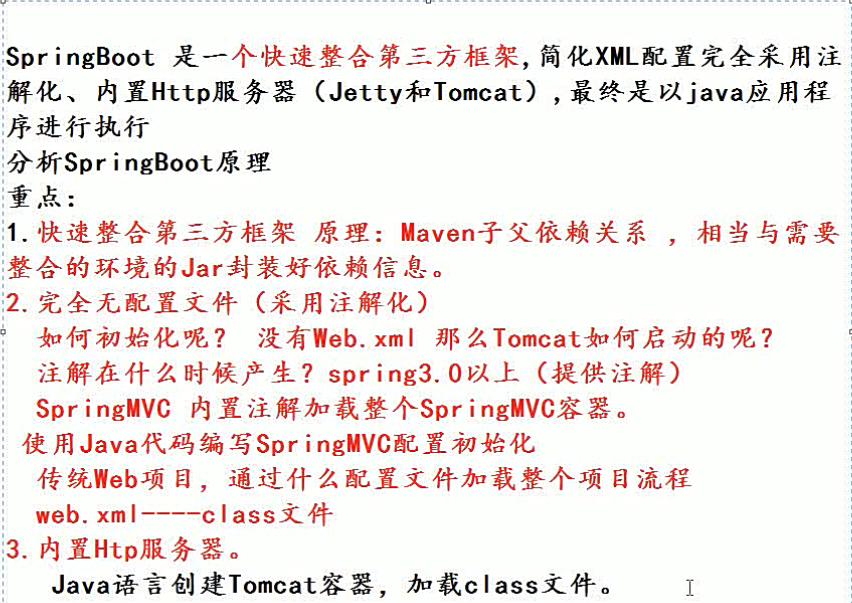




1. **Springboot监控中心可视化界面admin ui**



1. **Springboot运行原理**



1. **模拟Springboot内置tomcat服务器，处理http请求**
2. **引入tomcat依赖**
3. **创建servlet处理http请求**
4. **创建java启动类，创建tomcat，配置tomcat端口、上下文，添加servlet，配置请求映射**
5. **启动tomcat，在浏览器发出请求**
6. **纯手写Springboot**



