20240326淘天一面

项目
    南海系统
    中英文分词
    在面对高流量时，es如何处理
八股
    mysql底层数据结构
    联合索引最左前缀原则
    springboot自动配置原理
        总结
    springboot配置方式
    spring依赖注入
    Spring是怎么解决循环依赖的？
    spring mvc 工作原理
    AOP
    服务器FullGC怎么排查
    缓存
    消息队列

项目

南海系统

1. 对于服务器上传任意文件进行攻击，如何去避免。
2. 系统的使用用户
3. 登录是怎么做的？spring security
4. 表结构设计
5. es倒排索引

中英文分词

采用的是IK Analyzer分词器

```
@IndexField(fieldType = FieldType.TEXT, analyzer = Analyzer.IK_SMART, searchAnalyzer = Analyzer.IK_MAX_WORD)
    private String content;
```

IK Analyzer 分词示例：

- 输入文本内容：`"数据库索引可以大幅提高查询速度"`
- 分词结果：
    - 智能模式IK_SMART：`[数据库,索引,可以,大幅,提高,查询,速度]`
    - 细粒度切分模式IK_MAX_WORD：`[数据库,数据,索引,可以,大幅,提高,查询,速度]`

在面对高流量时，es如何处理

八股

mysql底层数据结构

联合索引最左前缀原则

springboot自动配置原理

https://juejin.cn/post/7046554366068654094

Spring的自动装配（Autowiring）是指Spring容器自动地将应用程序中的组件（如Bean）按照某种规则连接起来的过程。这个过程不需要开发者手动编写代码来创建和连接这些组件，而是通过Spring的依赖注入（Dependency Injection）机制来实现。

自动装配的内容包括：

1. **Bean的创建**：Spring容器负责创建Bean实例，这些Bean可能是通过`@Configuration`类中的`@Bean`注解方法创建的，也可能是通过其他方式（如Java注解或XML配置）定义的。
2. **依赖注入**：当一个Bean需要其他Bean作为其属性或依赖时，Spring容器会自动将所需的Bean注入到当前Bean中。这可以通过构造器注入、Setter注入、字段注入等方式实现。
3. **Bean的连接**：Spring容器会根据Bean之间的依赖关系将它们连接起来。例如，如果一个Bean A需要一个实现了特定接口的Bean B，Spring容器会自动找到并注入一个匹配的Bean B实例到Bean A中。
4. **配置类和组件的装配**：通过`@ComponentScan`注解，Spring容器会自动扫描指定包路径下的组件（如`@Service`、`@Repository`、`@Controller`等），并将它们注册为Bean。
5. **自动配置**：Spring Boot中的自动配置机制会根据项目中存在的依赖和配置，自动创建和配置一些常用的Bean。例如，如果项目中包含了`spring-boot-starter-web`依赖，Spring Boot会自动配置一个Web应用所需的各种Bean，如Servlet容器、MVC配置等。

自动装配的目的是为了简化配置，减少开发者的工作量，使得开发者可以更专注于业务逻辑的实现。通过自动装配，Spring容器能够智能地管理Bean之间的依赖关系，自动完成组件的装配工作，从而提高开发效率和代码的可维护性。没有 Spring Boot 的情况下，如果我们需要引入第三方依赖，需要手动配置，非常麻烦。但是，Spring Boot 中，我们直接引入一个 starter 即可。引入 starter 之后，我们通过少量注解和一些简单的配置就能使用第三方组件提供的功能了。

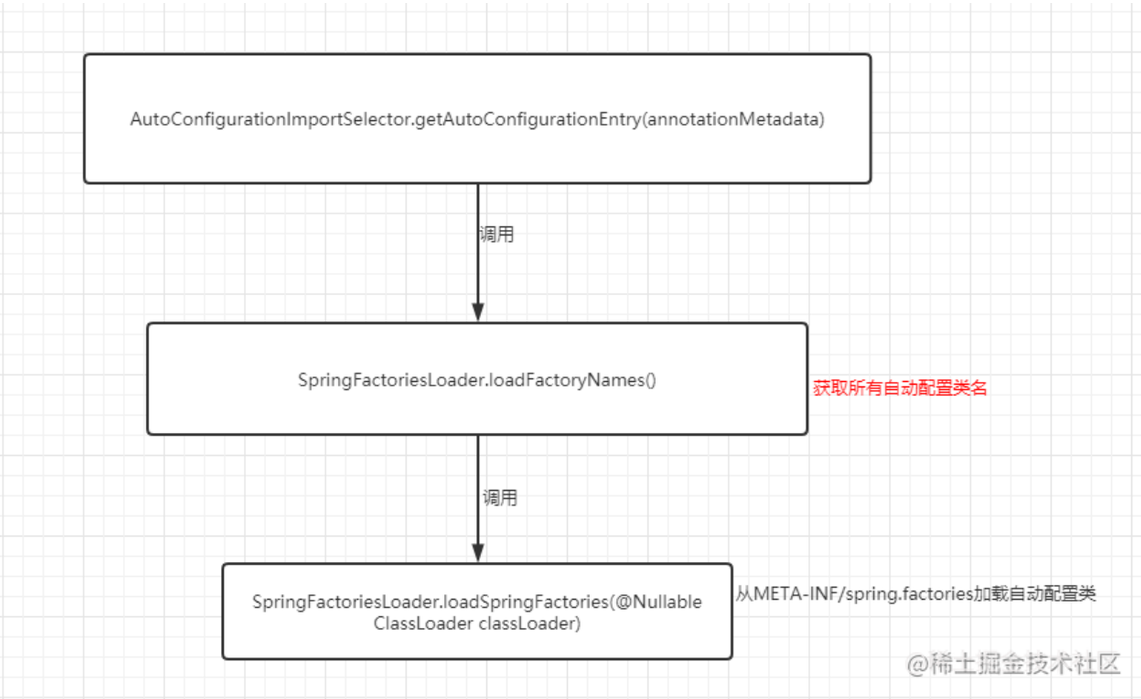
实现细节：

@EnableAutoConfiguration注解中包含：

1. @AutoConfigurationPackage：将main包下的所有组件注册到容器中
2. @Import({AutoConfigurationImportSelector.class})：加载自动装配类xxxAutoconfiguration

AutoConfigurationImportSelector类实现了ImportSelector接口，也就实现了这个接口中的selectImports方法，该方法主要用于**获取所有符合条件的类的全限定类名，这些类需要被加载到 IoC 容器中。**

selectImports方法中调用了`getAutoConfigurationEntry()`方法，这个方法主要负责加载自动配置类的。



不光是这个依赖下的`META-INF/spring.factories`被读取到，所有 Spring Boot Starter 下的`META-INF/spring.factories`都会被读取到。然后会通过`@ConditionalOnXXX`注解进行过滤，只有`@ConditionalOnXXX`中的所有条件都满足，该类才会生效。

总结

Spring Boot的自动装配原理是通过`@EnableAutoConfiguration`注解启用的，该注解利用`AutoConfigurationImportSelector`类来实现。这个类会扫描项目依赖中的`META-INF/spring.factories`文件，从中加载配置的自动配置类（自动配置类其实就是通过`@Conditional`按需加载的配置类，想要其生效必须引入`spring-boot-starter-xxx`包实现起步依赖），并结合条件注解（如`@ConditionalOnClass`、`@ConditionalOnBean`等）来按需加载相应的配置，从而实现在不需要显式XML或Java配置的情况下自动装配Spring应用的组件。这种方式极大地简化了Spring应用的配置过程，提高了开发效率。

eg：自定义线程池配置类

这段代码是一个Spring Boot自动配置类的例子，它的作用是在满足特定条件时自动配置一个线程池（`ThreadPoolExecutor`）Bean。

1. `@Configuration`注解表明`ThreadPoolAutoConfiguration`类是一个配置类，Spring容器会为此类创建一个bean定义，并将其添加到应用上下文中。
2. `@Bean`注解表明方法`MyThreadPool`将返回一个对象，该对象将被注册为Spring应用上下文的一个Bean。在这个例子中，`MyThreadPool`方法将创建并返回一个`ThreadPoolExecutor`实例。
3. `@ConditionalOnClass(ThreadPoolExecutor.class)`是一个条件注解，它指定了自动配置的条件。当类路径下存在`ThreadPoolExecutor`类时，这个条件才会满足。由于`ThreadPoolExecutor`是JDK自带的类，所以这个条件总是会满足，这意味着`MyThreadPool`方法总是会被执行，从而创建线程池。

这段代码的作用是在Spring Boot应用中自动配置并提供一个具有特定参数的线程池，供应用中的其他组件使用。这样做的好处是开发者不需要手动创建和配置线程池，而是可以依赖Spring Boot的自动装配机制来完成这项工作。

springboot配置方式

也许通过一个springboot starter来配置？

spring依赖注入

https://www.51cto.com/article/716969.html

依赖注入的常见实现方式有 3 种：属性注入、Setter 注入和构造方法注入。其中属性注入的写法最简单，所以日常项目中使用的频率最高，但它的通用性不好；而 Spring 官方推荐的是构造方法注入，它可以注入不可变对象，其通用性也更好，如果是注入可变对象，那么可以考虑使用 Setter 注入。

属性注入

```
@RestController
public class UserController {
    // 属性对象
    @Autowired
    private UserService userService;
```

```
    @RequestMapping("/add")
    public UserInfo add(String username, String password) {
        return userService.add(username, password);
    }
}
```

Setter注入

```
@RestController
public class UserController {
    // Setter 注入
    private UserService userService;

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @RequestMapping("/add")
    public UserInfo add(String username, String password) {
        return userService.add(username, password);
    }
}
```

构造方法注入

```
@RestController
public class UserController {
    // 构造方法注入
    private UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @RequestMapping("/add")
    public UserInfo add(String username, String password) {
        return userService.add(username, password);
    }
}
```

Spring是怎么解决循环依赖的？

首先，需要明确的是spring对循环依赖的处理有三种情况：

1. 构造器的循环依赖：这种依赖spring是处理不了的，直接抛BeanCurrentlyInCreationException异常。

2. 单例模式下的setter循环依赖：通过“三级缓存”处理循环依赖。
3. 非单例循环依赖：无法处理。

接下来，我们具体看看spring是如何处理第二种循环依赖的。

Spring单例对象的初始化大略分为三步：

1. createBeanInstance：实例化，其实也就是调用对象的构造方法实例化对象；
2. populateBean：填充属性，这一步主要是多bean的依赖属性进行填充；
3. initializeBean：调用spring xml中的init 方法。

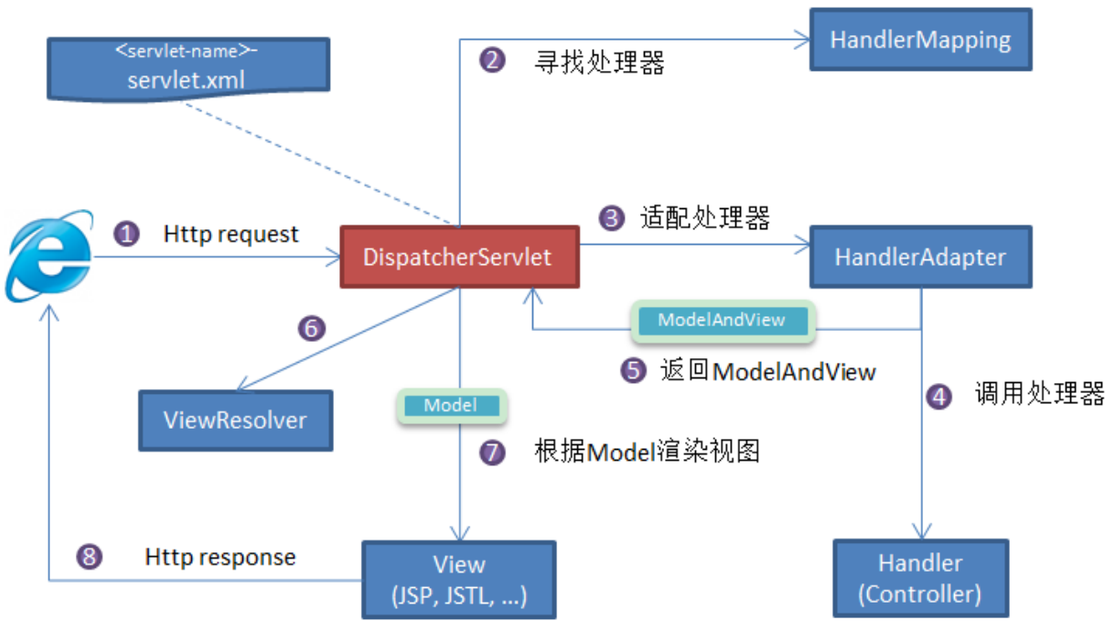
从上面讲述的单例bean初始化步骤我们可以知道，循环依赖主要发生在第一步、第二步。也就是构造器循环依赖和field循环依赖。 Spring为了解决单例的循环依赖问题，使用了三级缓存。

这三级缓存的作用分别是：

- singletonFactories ： 进入实例化阶段的单例对象工厂的cache （三级缓存）；
- earlySingletonObjects ：完成实例化但是尚未初始化的，提前暴光的单例对象的Cache （二级缓存）；
- singletonObjects：完成初始化的单例对象的cache（一级缓存）。

这样做有什么好处呢？让我们来分析一下“A的某个field或者setter依赖了B的实例对象，同时B的某个field或者setter依赖了A的实例对象”这种循环依赖的情况。A首先完成了初始化的第一步，并且将自己提前曝光到singletonFactories中，此时进行初始化的第二步，发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程，B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects（也没有），尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象(虽然A还没有初始化完全，但是总比没有好呀)，B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中。此时返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，进去了一级缓存singletonObjects中，而且更加幸运的是，由于B拿到了A的对象引用，所以B现在hold住的A对象完成了初始化。

spring mvc 工作原理



1. 客户端（浏览器）发送请求， `DispatcherServlet`拦截请求。
2. `DispatcherServlet` 根据请求信息调用 `HandlerMapping` 。`HandlerMapping` 根据 URL 去匹配查找能处理的 `Handler` （也就是我们平常说的 `Controller` 控制器） ，并会将请求涉及到的拦截器和 `Handler` 一起封装。
3. `DispatcherServlet` 调用 `HandlerAdapter`适配器执行 `Handler` 。
4. `Handler` 完成对用户请求的处理后，会返回一个 `ModelAndView` 对象给 `DispatcherServlet`，`ModelAndView` 顾名思义，包含了数据模型以及相应的视图的信息。`Model` 是返回的数据对象，`View` 是个逻辑上的 `View`。
5. `ViewResolver` 会根据逻辑 `View` 查找实际的 `View`。
6. `DispaterServlet` 把返回的 `Model` 传给 `View` （视图渲染）。
7. 把 `View` 返回给请求者（浏览器）

AOP

有哪几种实现方式？

Spring AOP（运行时织入）

https://www.cnblogs.com/tuyang1129/p/12878549.html

`Spring`的`AOP`实现原理其实很简单，就是通过动态代理实现的。如果我们为`Spring`的某个`bean`配置了切面，那么`Spring`在创建这个`bean`的时候，实际上创建的是这个`bean`的一个代理对象，我们后续对`bean`中方法的调用，实际上调用的是代理类重写的代理方法。而`Spring`的`AOP`使用了两种动态代理，分别是JDK的动态代理，以及CGLib的动态代理。

Spring默认使用JDK的动态代理实现AOP，类如果实现了接口，Spring就会使用这种方式实现动态代理。`JDK`的动态代理存在限制，那就是被代理的类必须是一个实现了接口的类，代理类需要实现相同的接口，代理接口中声明的方法。若需要代理的类没有实现接口，此时`JDK`的动态代理将没有办法使用，于是`Spring`会使用`CGLib`的动态代理来生成代理对象。`CGLib`直接操作字节码，生成类的子类，重写类的方法完成代理。

JDK动态代理在内部使用反射机制来拦截代理对象的方法调用，通过实现`InvocationHandler`接口的`invoke`方法，可以拦截并自定义代理对象的方法调用。CGLIB通

过字节码技术在运行时生成一个新的子类，来实现对目标类的代理。与JDK动态代理类似，CGLIB也提供了方法拦截的功能，但需要使用`MethodInterceptor`接口。

Aspectj（编译or类加载时织入）

一种就是我们常见的**基于java注解**切面描述的方法，这种方法兼容java语法，写起来十分方便，不需要IDE的额外语法检测支持；另外一种是**基于aspect文件**的切面描述方法，这种语法本身并不是java语法，因此写的时候需要IDE的插件支持才能进行语法检查。

三种织入方式：

https://www.javadoop.com/post/aspectj

1. compile-time：这是最简单的使用方式，在编译期的时候进行织入（即在编译的时候先修改了代码再进行编译），这样编译出来的 .class 文件已经织入了我们的代码，在 JVM 运行的时候其实就是加载了一个普通的被织入了代码的类。可以运用`aspectj-maven-plugin`插件来实现编译。
2. post-compile：编译后织入，增强已经编译出来的类，如我们要增强依赖的 jar 包中的某个类的某个方法。可以运用`aspectj-maven-plugin`插件来实现编译。
3. load-time：在 JVM 进行类加载的时候进行织入。这种方式不需要使用AspectJ编译器，而是在类加载到JVM时，通过自定义的类加载器来实现。AspectJ的weaver会在类加载时读取类文件，应用切点和通知，然后生成新的类文件来替换原始的类。

Java agent（更通用的字节码增强方式）

Java Agent是通过`-javaagent`参数在JVM启动时或运行时附加到JVM上的一个组件。它使用JVM Tool Interface (JVMTI)来监听和修改JVM的行为。Java Agent可以在以下时机进行类文件的修改：

- **类加载前premain**：在类文件被加载到JVM之前，可以拦截并修改字节码。通过在 java 的启动参数中添加 -javaagent:/jar包路径来进行字节码增强，随着java进程启动而启动。
- **类加载后agentmain**：对于已经加载到JVM中的类，可以在其被重新定义（redefine）时修改字节码。在运行时通过JVM Attach机制进行字节码增强。

Jvm SandBox

服务器FullGC怎么排查

https://juejin.cn/post/6957903936657293319#heading-40

https://heapdump.cn/article/1870333

下面4种情况，对象会进入到老年代中：

- YGC时，To Survivor区不足以存放存活的对象，对象会直接进入到老年代。
- 经过多次YGC后，如果存活对象的年龄达到了设定阈值，则会晋升到老年代中。
- 动态年龄判定规则，To Survivor区中相同年龄的对象，如果其大小之和占到了 To Survivor区一半以上的空间，那么大于此年龄的对象会直接进入老年代，而不需要达到默认的分代年龄。
- 大对象：由-XX:PretenureSizeThreshold启动参数控制，若对象大小大于此值，就会绕过新生代, 直接在老年代中分配。

当晋升到老年代的对象大于了老年代的剩余空间时，就会触发FGC（Major GC），FGC处理的区域同时包括新生代和老年代。除此之外，还有以下4种情况也会触发FGC：

- 老年代的内存使用率达到了一定阈值（可通过参数调整），直接触发FGC。
- 空间分配担保：在YGC之前，会先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间。如果小于，说明YGC是不安全的，则会查看参数HandlePromotionFailure 是否被设置成了允许担保失败，如果不允许则直接触发Full GC；如果允许，那么会进一步检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果小于也会触发 Full GC。
- Metaspace（元空间）在空间不足时会进行扩容，当扩容到了-XX:MetaspaceSize参数的指定值时，也会触发FGC。
- System.gc() 或者Runtime.gc() 被显式调用时，触发FGC。

从程序角度，有哪些原因导致FGC？

- 大对象：系统一次性加载了过多数据到内存中（比如SQL查询未做分页），导致大对象进入了老年代。
- 内存泄漏：频繁创建了大量对象，但是无法被回收（比如IO对象使用完后未调用close方法释放资源），先引发FGC，最后导致OOM.
- 程序频繁生成一些长生命周期的对象，当这些对象的存活年龄超过分代年龄时便会进入老年代，最后引发FGC.
- 程序BUG导致动态生成了很多新类，使得 Metaspace 不断被占用，先引发FGC，最后导致OOM.
- 代码中显式调用了gc方法，包括自己的代码甚至框架中的代码。
- JVM参数设置问题：包括总内存大小、新生代和老年代的大小、Eden区和S区的大小、元空间大小、垃圾回收算法等等。

排查指南

1. 了解JVM的参数设置，包括：堆空间各个区域的大小设置，新生代和老年代分别采用了哪些垃圾收集器，然后分析JVM参数设置是否合理。
2. 元空间被打满、内存泄漏、代码显式调用gc方法比较容易排查。
3. 针对大对象或者长生命周期对象导致的FGC，可通过 jmap -histo 命令并结合dump堆内存文件作进一步分析，需要先定位到可疑对象。
4. 通过可疑对象定位到具体代码再次分析，这时候要结合GC原理和JVM参数设置，弄清楚可疑对象是否满足了进入到老年代的条件才能下结论。

缓存

redis 缓存击穿

消息队列