【这节课的主要内容是基于代码的演示和讲解，文字性的东西比较少，大家主要是去理解课程中写的代码】

什么是 Starter

Starter 是 Spring Boot 中的一个非常重要的概念，Starter 相当于模块，它能将模块所需的依赖整合起来并对模块内的 Bean 根据环境（ 条件）进行自动配置。使用者只需要依赖相应功能的 Starter，无需做过多的配置和依赖，Spring Boot 就能自动扫描并加载相应的模块。
在上一节课中，我们在 Maven 的依赖中加入 spring-boot-starter-web 就能使项目支持 Spring MVC，并且 Spring Boot 还为我们做了很多默认配置，无需再依赖 spring-web、spring-webmvc 等相关包及做相关配置就能够立即使用起来

SpringBoot 存在很多开箱即用的 Starter 依赖，使得我们在开发业务代码时能够非常方便的、不需要过多关注框架的配置，而只需要关注业务即可

spring-boot-starter-logging

在实际应用中，日志是最重要的一个组件：

1. 它可以为系统提供错误以及日常的定位；
2. 也可以对访问的记录进行跟踪；
3. 当然，在很多大型的互联网应用中，基于日志的收集以及分析可以了解用户的用户画像，比如兴趣爱好、点击行为。

常见的日志框架

可能是太过于常见了，所以使得大家很少关注，只是要用到的时候复制粘贴一份就行，甚至连日志配置文件中的配置语法都不清楚。另外一方面，Java 中提供的日志组件太多了，一会儿 log4j，一会儿 logback，一会儿又是 log4j2. 不清楚其中的关联

Java 中常用的日志框架： Log4j、Log4j 2、Commons Logging、Slf4j、Logback、Jul(Java Util Logging)

简单介绍日志的发展历史

最早的日志组件是 Apache 基金会提供的 Log4j，log4j 能够通过配置文件轻松的实现日志系统的管理和多样化配置，所以很快被广泛运用。也是我们接触得比较早和比较多的日志组件。它几乎成了 Java 社区的日志标准。

据说 Apache 基金会还曾经建议 Sun 引入 Log4j 到 java 的标准库中，但 Sun 拒绝了。 所以 sun 公司在 java1.4 版本

中，增加了日志库(Java Util Logging)。其实现基本模仿了 Log4j 的实现。在 JUL 出来以前，Log4j 就已经成为一项成熟的技术，使得 Log4j 在选择上占据了一定的优势

Apache 推出的 JUL 后，有一些项目使用 JUL，也有一些项目使用 log4j，这样就造成了开发者的混乱，因为这两个日志组件没有关联，所以要想实现统一管理或者替换就非常困难。怎么办呢?

这个状况交给你来想想办法，你该如何解决呢？进行抽象，抽象出一个接口层，对每个日志实现都适配，这样这些提供给别人的库都直接使用抽象层即可

这个时候又轮到 Apache 出手了，它推出了一个 Apache Commons Logging 组件，JCL 只是定义了一套日志接口(其内部也提供一个 Simple Log 的简单实现)，支持运行时动态加载日志组件的实现，也就是说，在你应用代码里，只需调用 Commons Logging 的接口，底层实现可以是 Log4j，也可以是 Java Util Logging

由于它很出色的完成了主流日志的兼容，所以基本上在后面很长一段时间，是无敌的存在。连 spring 也都是依赖 JCL 进行日志管理

但是故事并没有结束

原 Log4J 的作者，它觉得 Apache Commons Logging 不够优秀，所以他想搞一套更优雅的方案，于是 slf4j 日志体

系诞生了，slf4j 实际上就是一个日志门面接口，它的作用类似于 Commons Loggins。 并且他还为 slf4j 提供了一个日志的实现-logback。

因此大家可以发现 Java 的日志领域被划分为两个大营：Commons Logging 和 slf4j

另外，还有一个 log4j2 是怎么回事呢？ 因为 slf4j 以及它的实现 logback 出来以后，很快就赶超了原本 apache 的 log4j 体系，所以 apache 在 2012 年重写了 log4j， 成立了新的项目 Log4j2

总的来说，日志的整个体系分为日志框架和日志系统

日志框架：JCL/ Slf4j

日志系统：Log4j、Log4j2、Logback、JUL。

而在我们现在的应用中，绝大部分都是使用 slf4j 作为门面，然后搭配 logback 或者 log4j2 日志系统

SpringBoot 另一大神器-Actuator

微服务应用开发完成以后，最终目的是为了发布到生产环境上给用户试用，开发结束并不意味着研发的生命周期结束，更多的时候他只是一个开始，因为服务在本地测试完成以后，并不一定能够非常完善的考虑到各种场景。所以需要通过运维来保障服务的稳定。

在以前的传统应用中，我们可以靠人工来监控。但是微服

务中，几千上万个服务，我们需要了解每个服务的健康状态，就必须要依靠监控平台来实现。
所以在 SpringBoot 框架中提供了 spring-boot-starter-actuator 自动配置模块来支持对于 SpringBoot 应用的监控

Actuator

Spring Boot Actuator 的关键特性是在应用程序里提供众多 Web 端点，通过它们了解应用程序运行时的内部状况。
有了 Actuator，你可以知道 Bean 在 Spring 应用程序上下文里是如何组装在一起的，掌握应用程序可以获取的环境属性信息
在 spring-boot 项目中，添加 actuator 的一个 starter.

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

Actuator 提供的 endpoint

启动服务之后，可以通过下面这个地址看到 actuator 提供的所有 Endpoint 地址

http://localhost:8080/actuator

可以看到非常多的 Endpoint。 有一些 Endpoint 是不能访问的，涉及到安全问题。

如果想开启访问那些安全相关的 url，可以在 application.xml 中配置， 开启所有的 endpoint

```
management.endpoints.web.exposure.include=*
```

health

针对当前 SpringBoot 应用的健康检查，默认情况下，会通过“up”或者“down”；可以基于下面这个配置，来打印 heath 更详细的信息

```
management.endpoint.health.show-details=always
```

Loggers

显示当前 spring-boot 应用中的日志配置信息，针对每个 package 对应的日志级别

beans

获取当前 spring-boot 应用中 IoC 容器中所有的 bean

Dump

获取活动线程的快照

Mappings

返回全部的 uri 路径，以及和控制器的映射关系

conditions

显示当前所有的条件注解，提供一份自动配置生效的条件

情况，记录哪些自动配置条件通过了，哪些没通过

shutdown

关闭应用程序，需要添加这个配置：

```
management.endpoint.shutdown.enabled=true
```

这个 Endpoint 是比较危险的，如果没有一定的安全保障，不要开启

Env

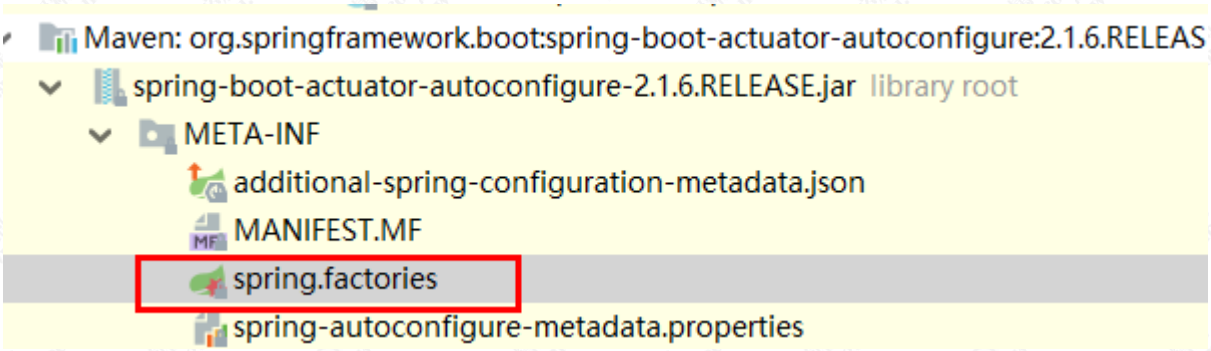
获取全部的环境信息

关于 health 的原理

应用健康状态的检查应该是监控系统中最基本的需求，所以我们基于 health 来分析一下它是如何实现的。

SpringBoot 预先通过 org.springframework.boot.actuate.autoconfigure.health.HealthIndicatorAutoConfiguration 这个就是基于 spring-boot 的自动装配来载入的。

所以，我们可以在 actuator-autoconfigure 这个包下找到 spring.factories。



Actuator 中提供了非常多的扩展点，默认情况下提供了一些常见的服务的监控检查的支持。

DataSourceHealthIndicator

DiskSpaceHealthIndicator

RedisHealthIndicator

...

其中，有一些服务的检查，需要依赖于当前应用是否集成了对应的组件，比如 redis，如果没有集成，那么 RedisHealthIndicatorAutoConfiguration 就不会被装载。因为它有 condition 的条件判断

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.actuate.autoconfigure.amqp.RabbitHealthInd
org.springframework.boot.actuate.autoconfigure.audit.AuditAutoConfi
org.springframework.boot.actuate.autoconfigure.audit.AuditEventsEnd
org.springframework.boot.actuate.autoconfigure.beans.BeansEndpointA
org.springframework.boot.actuate.autoconfigure.cache.CachesEndpoint
org.springframework.boot.actuate.autoconfigure.cassandra.CassandraH
org.springframework.boot.actuate.autoconfigure.cassandra.CassandraR
org.springframework.boot.actuate.autoconfigure.cloudfoundry.servlet
org.springframework.boot.actuate.autoconfigure.cloudfoundry.reactiv
org.springframework.boot.actuate.autoconfigure.condition.Conditions
org.springframework.boot.actuate.autoconfigure.context.properties.C
org.springframework.boot.actuate.autoconfigure.context.ShutdownEndp
org.springframework.boot.actuate.autoconfigure.couchbase.CouchbaseH
org.springframework.boot.actuate.autoconfigure.couchbase.CouchbaseR
org.springframework.boot.actuate.autoconfigure.elasticsearch.Elasti
org.springframework.boot.actuate.autoconfigure.elasticsearch.Elasti
org.springframework.boot.actuate.autoconfigure.elasticsearch.Elasti
```

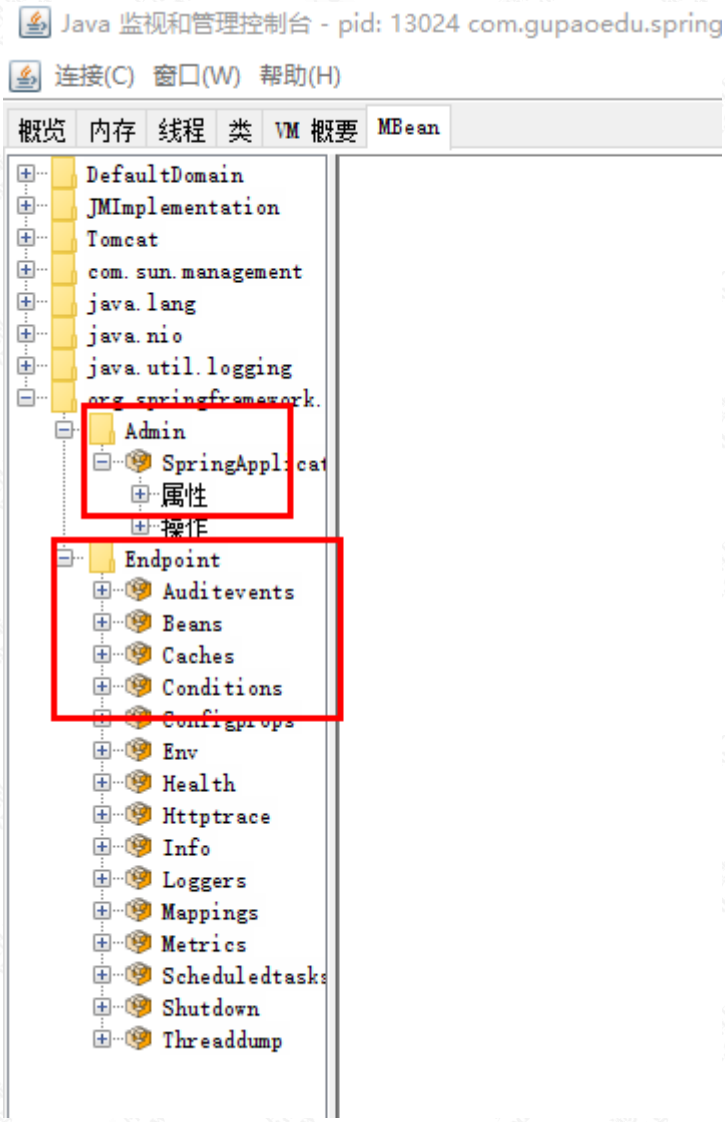
Actuator 对于 JMX 支持

除了 REST 方式发布的 Endpoint，Actuator 还把它的端点以 JMX MBean 的方式发布出来，可以通过 JMX 来查看和管理。

操作步骤

在 cmd 中输入 jconsole，连接到 spring-boot 的应用

就可以看到 JBean 的信息以及相应的操作。比如可以在操作菜单中访问 shutdown 的 endpoint 来关闭服务



什么是 JMX

JMX 全称是 Java Management Extensions。 Java 管理扩展。它提供了对 Java 应用程序和 JVM 的监控和管理功能。

通过 JMX，我们可以监控

1. 服务器中的各种资源的使用情况，CPU、内存
2. JVM 内存的使用情况
3. JVM 线程使用情况

比如前面讲的 Actuator 中，就是基于 JMX 的技术来实现

对 endpoint 的访问