Spring Boot

应用手册

V1.0

**目 录**

[1 简介 1](#_Toc488932920)

[1.1 Spring Boot是什么 1](#_Toc488932921)

[1.2 Spring Boot的特点 1](#_Toc488932922)

[1.3 Spring Boot解决的问题 2](#_Toc488932923)

[1.3.1 Spring Boot使编码变简单 2](#_Toc488932924)

[1.3.2 Spring Boot使配置变简单 3](#_Toc488932925)

[1.3.3 Spring Boot使部署变简单 4](#_Toc488932926)

[1.3.4 Spring Boot使监控变简单 4](#_Toc488932927)

[1.3.5 Spring Boot的不足 5](#_Toc488932928)

[1.4 Spring Boot的学习成本 5](#_Toc488932929)

[1.5 Spring Boot周围的工具链 6](#_Toc488932930)

[2 Spring Boot的功能 6](#_Toc488932931)

[2.1 独立运行的Spring项目 6](#_Toc488932932)

[2.2 内嵌的Servlet容器 6](#_Toc488932933)

[2.3 提供starter，简化Manen配置 7](#_Toc488932934)

[2.4 自动配置Spring 7](#_Toc488932935)

[2.5 应用监控 8](#_Toc488932936)

[2.6 无代码生成和XML配置 10](#_Toc488932937)

[2.7 相关技术的融合 10](#_Toc488932938)

[2.7.1 Spring Boot与SEDA + MicroService + REST 11](#_Toc488932939)

[2.7.2 Spring Boot与Mock 11](#_Toc488932940)

[2.7.3 Spring Boot与Docker 12](#_Toc488932941)

[2.8 运行期安全管理 13](#_Toc488932942)

[3 使用Eclipse开发Spring Boot工程 14](#_Toc488932943)

[3.1 创建Maven工程 14](#_Toc488932944)

[3.2 修改pom.xml 18](#_Toc488932945)

[3.3 测试Spring Boot应用 21](#_Toc488932946)

[4 Spring Boot使用 23](#_Toc488932947)

[4.1 Spring Boot初探 23](#_Toc488932948)

[4.2 数据库访问 24](#_Toc488932949)

[4.2.1 使用JPA 24](#_Toc488932950)

[4.2.2 使用JdbcTemplate 28](#_Toc488932951)

[4.2.3 使用Mybatis 30](#_Toc488932952)

[4.3 配置 37](#_Toc488932953)

[4.4 @EnableAutoCongiguration 39](#_Toc488932954)

[4.5 Production特性 42](#_Toc488932955)

[4.6 通过Actuator对系统进行监控 42](#_Toc488932956)

[4.6.1 使用actuator 42](#_Toc488932957)

[4.6.2 主要暴露的功能 43](#_Toc488932958)

[4.6.2.1 返回数据示例：/autoconfig 43](#_Toc488932959)

[4.6.2.2 返回数据示例：/configprops 44](#_Toc488932960)

[4.6.2.3 返回数据示例：/beans 44](#_Toc488932961)

[4.6.2.4 返回数据示例：/dump 45](#_Toc488932962)

[4.6.2.5 返回数据示例：/env 47](#_Toc488932963)

[4.6.2.6 返回数据示例：/health 51](#_Toc488932964)

[4.6.2.7 返回数据示例：/info 51](#_Toc488932965)

[4.6.2.8 返回数据示例：/mappings 51](#_Toc488932966)

[4.6.2.9 返回数据示例：/metrics 52](#_Toc488932967)

[4.6.2.10 返回数据示例：/shutdown 54](#_Toc488932968)

[4.6.2.11 返回数据示例：/trace 54](#_Toc488932969)

[5 总结 55](#_Toc488932970)

# 简介

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

官方地址：[**http://projects.spring.io/spring-boot/**](http://projects.spring.io/spring-boot/)

## Spring Boot是什么

首先，我们来看一下**spring boot是什么，它帮助我们解决了哪些问题**：



Spring Boot继承了Spring的优点，并新增了一些新功能和特性：

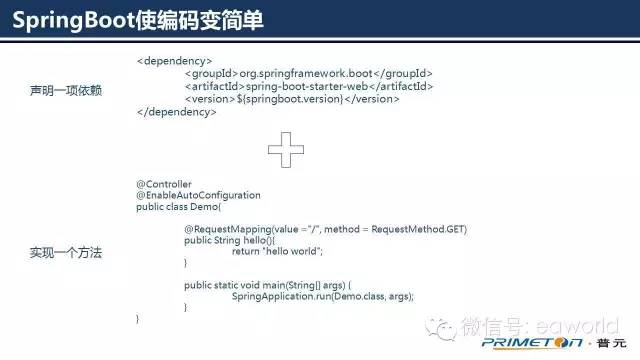
* Spring Boot是伴随着Spring4.0诞生的，一经推出，引起了巨大的反向；
* 从字面理解，Boot是引导的意思，因此Spring Boot帮助开发者快速搭建Spring框架；
* Spring Boot帮助开发者快速启动一个Web容器；
* Spring Boot继承了原有Spring框架的优秀基因；
* Spring Boot简化了使用Spring的过程；
* Spring Boot为我们带来了脚本语言开发的效率，但是Spring Boot并没有让我们意外的新技术，都是Java EE开发者常见的技术。

## Spring Boot的特点

* 创建独立的Spring应用程序。
* 遵循“习惯优于配置”的原则，使用Spring Boot只需要很少的配置，大部分的时候我们直接使用默认的配置即可；
* 项目快速搭建，简化Maven配置，可以无需配置的自动整合第三方的框架；
* 绝对没有代码生成和对XML没有要求配置，可以完全不使用XML配置文件，只需要自动配置和Java Config；
* 内嵌Servlet容器（tomcat，可更换为Jetty），降低了对环境的要求，，无需部署WAR文件,可以使用命令直接执行项目，应用可用jar包执行：java -jar；
* 提供了starter POM, 能够非常方便的进行包管理, 很大程度上减少了jar hell或者dependency hell；
* 运行中应用状态的监控，提供生产就绪型功能，如指标，健康检查和外部配置；
* 对主流开发框架的无配置集成；
* 对云计算的天然支持；

## Spring Boot解决的问题

### Spring Boot使编码变简单



传统基于Spring的Java Web应用，需要配置web.xml, applicationContext.xml，将应用打成war包放入应用服务器(Tomcat, Jetty等)中并运行。如果基于Spring Boot，这一切都将变得简单：

在pom.xml中声明一项依赖：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>${springboot.version}</version>

</dependency>

实现一个方法：

@Controller

@EnableAutoConfiguration

public class SampleController {

@RequestMapping("/")

@ResponseBody

String home() {

return "Hello World!";

}

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

SpringApplication.run(SampleController.class, args);

}

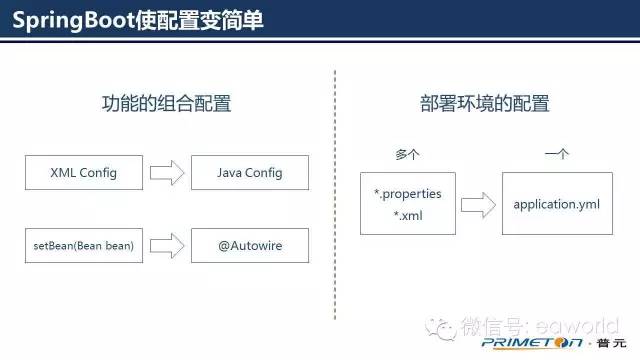
}

启动main函数后，在控制台中可以发现启动了一个Tomcat容器，一个基于Spring MVC的应用也同时启动起来，这时访问[http://localhost:8080](http://localhost:8080/)就可以看到Hello World!出现在浏览器中了。

### Spring Boot使配置变简单

Spring由于其繁琐的配置，一度被人认为**“配置地狱”**，各种XML、Annotation配置，让人眼花缭乱，而且如果出错了也很难找出原因。

Spring Boot更多的是采用Java Config的方式，对Spring进行配置。



### Spring Boot使部署变简单



* 一键启动：无需解压，直接通过 java –jar standalone.jar即可运行。
* 无需部署应用服务器：自带Java Web容器（tomcat，可换为Jetty）。
* 降低对运行环境的要求：环境变量中有JDK即可；应用全量更新。

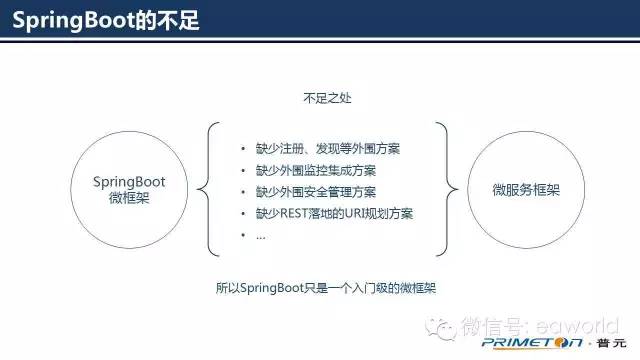
### Spring Boot使监控变简单

actuator是Spring Boot提供的对应用系统的自省和监控的集成功能，可以对应用系统进行配置查看、相关功能统计等。

| **HTTP方法** | **路径** | **描述** | **鉴权** |
| --- | --- | --- | --- |
| GET | /autoconfig | 查看自动配置的使用情况 | true |
| GET | /configprops | 查看配置属性，包括默认配置 | true |
| GET | /beans | 查看bean及其关系列表 | true |
| GET | /dump | 打印线程栈 | true |
| GET | /env | 查看所有环境变量 | true |
| GET | /env/{name} | 查看具体变量值 | true |
| GET | /health | 查看应用健康指标 | false |
| GET | /info | 查看应用信息 | false |
| GET | /mappings | 查看所有url映射 | true |
| GET | /metrics | 查看应用基本指标 | true |
| GET | /metrics/{name} | 查看具体指标 | true |
| POST | /shutdown | 关闭应用 | true |
| GET | /trace | 查看基本追踪信息 | true |

### Spring Boot的不足

可以看到，采用了spring-boot-start-actuator之后，直接以REST的方式，获取进程的运行期性能参数。当然这些metrics有些是有敏感数据的，spring-boot-start-actuator为此提供了一些Basic Authentication认证的方案，这些方案在实际应用过程中也是不足的。



Spring Boot作为一个微框架，离微服务的实现还是有距离的。没有提供相应的服务发现和注册的配套功能，自身的acturator所提供的监控功能，也需要与现有的监控对接。没有配套的安全管控方案，对于REST的落地，还需要自行结合实际进行URI的规范化工作。

## Spring Boot的学习成本

采用了Spring Boot之后，**技术管理应该如何进行**？



正因为Spring Boot是与Spring一脉相承的，所以对于广大的Java开发者而言，对于Spring的学习成本几乎为零。

在实践Spring Boot时学习重点，或者说**思维方式改变的重点在于**：

1）对于REST的理解，这一点尤为重要，需要从设计、开发多个角色达成共识，很多时候都是对于HTTP 1.1协议以及REST的精髓不理解，导致REST被「盲用」而产生一些不好的效果。

2）对于YAML的理解和对于JavaConfig的理解，这两点相对较为简单，本质上是简化了xml文件，并提供等价的配置表述能力。

## Spring Boot周围的工具链



丰富的工具链为SpringBoot的推广带来了利好。SpringBoot的工具链主要来自于两个方面：

* 原有Spring积累的工具链；
* SpringMVC或者其他REST框架使用HTTP协议，使得HTTP丰富的工具成为SpringBoot天然的资源。

# Spring Boot的功能

## 独立运行的Spring项目

Spring Boot可以以jar包的形式进行独立的运行，使用：java -jar xx.jar 就可以成功的运行项目，或者在应用项目的主程序中运行main函数即可；

## 内嵌的Servlet容器

内嵌容器，使得我们可以直接执行项目的主程序main函数，使项目快速运行。

示例：

package com.xuliugen.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringbootDemoApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootDemoApplication.class, args);

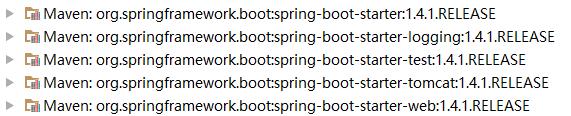
}

}

## 提供starter，简化Manen配置

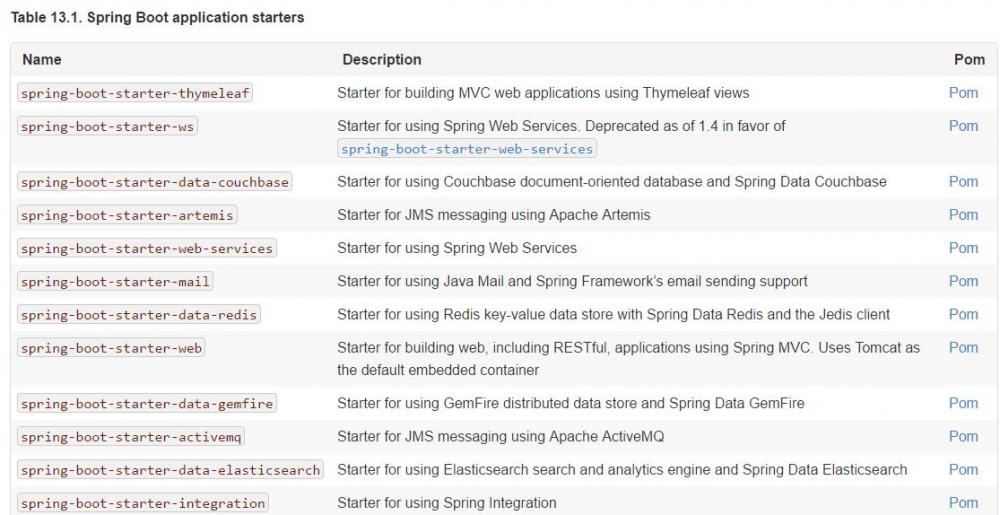
Spring Boot提供了一系列的starter pom用来简化我们的Maven依赖。

例如，使用spring-boot-starter-web依赖，系统自动关联以下内容：



Spring Boot官网还提供了很多的starter pom，请参考：

<http://docs.spring.io/spring-boot/docs/1.4.1.RELEASE/reference/htmlsingle/#using-boot-starter>



## 自动配置Spring

Spring Boot会根据我们项目中类路径的jar包或类，为jar包的类进行自动配置Bean，这样一来就大大的简化了我们的配置。

当然，这只是Spring考虑到的大多数的使用场景，在一些特殊情况，我们还需要自定义自动配置。

## 应用监控

Spring Boot提供了基于http、ssh、telnet对运行时的项目进行监控；这个是不是很棒！

示例：以SSH登录为例。

1、首先，添加starter pom依赖

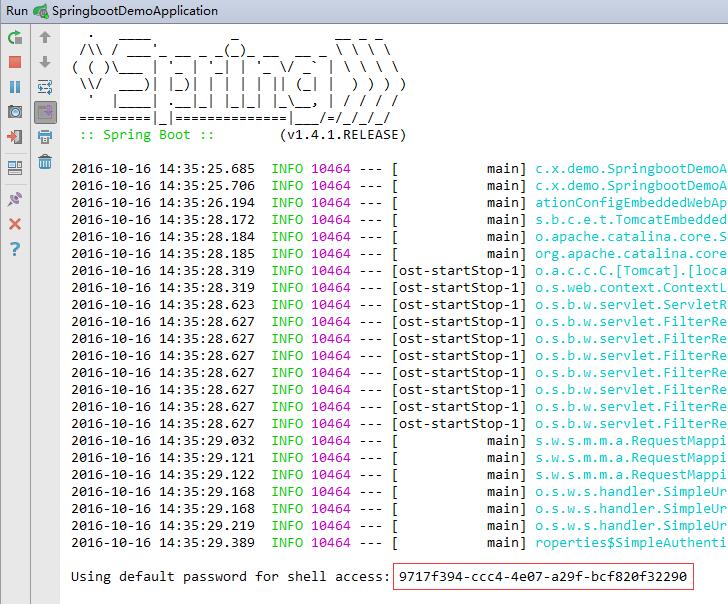
<dependency>

<groupId>org.springframework.boot</groupId>

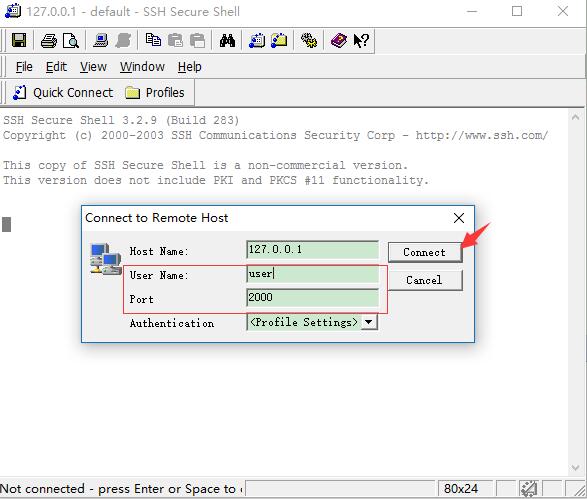
<artifactId>spring-boot-starter-remote-shell</artifactId>

</dependency>

2、运行项目,此时在控制台中会出现SSH访问的密码：



3、使用SecureCRT登录到我们的程序，端口为2000，用户为user：

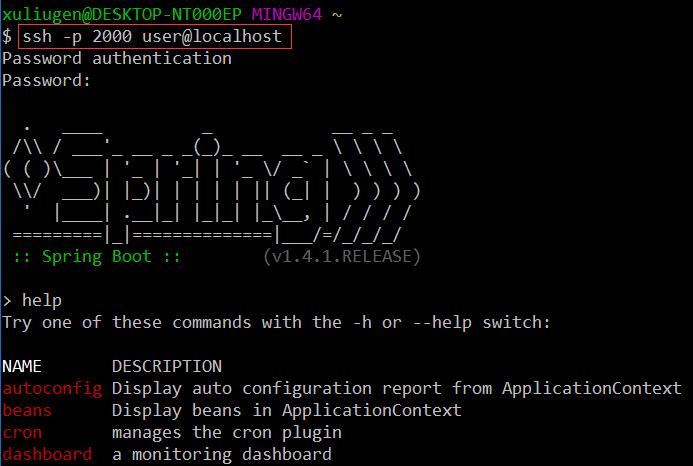


密码就是刚才的shell access；但是当我点击连接的时候，出现错误：



显然是SecureCRT的版本不支持，所以就放弃了这个，使用Git Bash：

ssh -p 2000 user@127.0.0.1



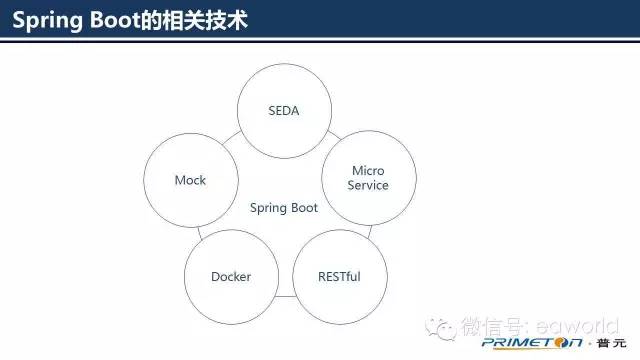
剩下的事情，大家自己玩吧！

## 无代码生成和XML配置

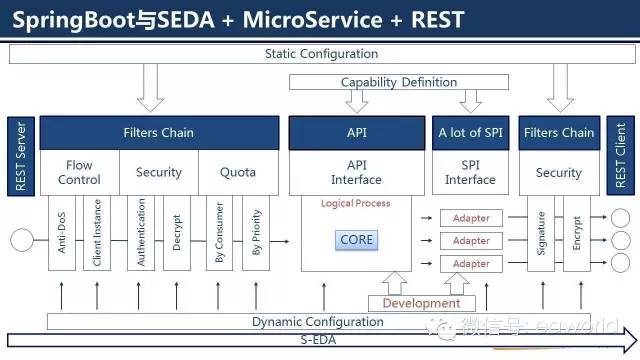
Spring Boot神奇的地方不是借助于代码生成来实现的，而是通过条件注解的方式来实现的，这也是Spring 4.x的新特性。

## 相关技术的融合

下面，我们研究一下**Spring Boot在平台中的定位，以及相关技术如何融合**。



### Spring Boot与SEDA + MicroService + REST



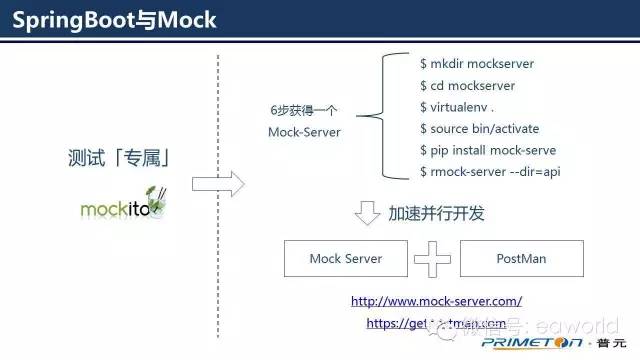
上图比较复杂，整体是采用SEDA，也就是Stage-EDA。可以看到，整体是以处理顺序进行展示的，响应过程类似。在处理过程中，主要会有前置过滤，核心功能处理，后置过滤几大部分。

图中的过滤器都是可插拔式的，并且可以根据实际场景进行扩展开发。每个过滤器都是Stage，比如ClientInstance合法性检查、调用鉴权、解密、限流等等。

一个请求Stage与Stage的转换，实现上是切换不同的线程池，并以EDA的方式驱动。

对于业务逻辑的开发者而言，只需要关心CORE部分的业务逻辑实现，其他的非功能都由框架进行统一实现。

### Spring Boot与Mock



Mock不应当再是测试的专有名词了（Mock的解释，参见百度百科：https://baike.baidu.com/item/mock%E6%B5%8B%E8%AF%95/5300937?fr=aladdin），当然对于测试这个角色而言，mockito这样的工具，依然可以为他们提升不少效率。

Spring Boot为创建REST服务提供了简便的途径，相比之下，采用阿里的dubbo在做多团队、多进程联调时，mock的难度就陡增。

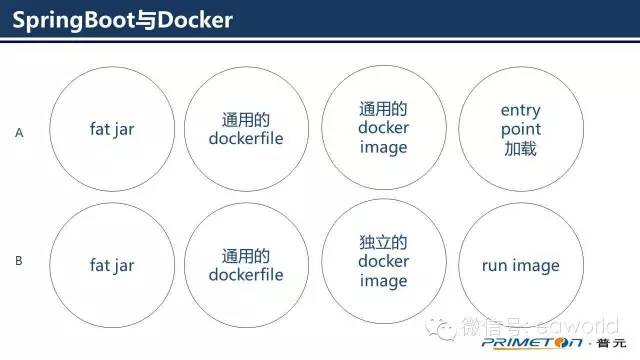
Mock是解耦并行开发的利器，在理性的情况下，软件从开发期Mock联调，到开发与开发的真实联调，只需要切换一个依赖的域名即可，比如：

mockURI:http://mock.service.net/v1/function?param1=value1

devURI:http://dev.service.net/v1/function?param1=value1

而上述的域名切换，只需要在开发期定义好一个配置项，在做环境切换的时候自动注入即可，省时、省心、省力。

### Spring Boot与Docker



如上图和**docker的集成可以有AB两种方案**：

* A方案的核心是，把docker作为操作系统环境的交付基线，也就是不同的fat jar 使用相同的操作系统版本、相同的JVM环境。但对于docker image来说都是一样的。
* B方案的核心是，不同的fat jar，独立的编译为docker image，在启动时直接启动带有特定版本的image。

A相比与B方案的特点是对于docker registry（也就是docker的镜像仓库）的依赖性较低，对于前期编译过程的要求也较低。

## 运行期安全管理



Spring Boot自身对于前面提到的配置文件：“application.yml”提供了多个「Profile」，可以便于开发者描述不同环境的配置，这些配置例如数据库的连接地址、用户名和密码。

但是对于企业用户而言，把不同环境的配置，写到同一个配置文件中，是极其不安全的，是一个非常危险的动作。

有一个经常被提及的例子是，随着开源的进行，很多互联网公司，都由于把相关的代码提交到github之类的开源代码社区，并且没有对代码进行严格的配置审查，导致一些“password”被公开。有些不良用心的人，就利用搜索工具，专门去挖掘这些关键字，进而导致数据库被「拖库」。

所以对于企业用户，更多的应该是采用集中式的配置管理系统，将不同环境的配置严格区分地存放。

虽然Spring Boot的actuator自身提供了基于「用户名+口令」的最简单的认证方式，但它保护的是对框架自身运行期的性能指标敏感数据的最基本的保护。这种保护在实际应用过程中，「用户名+口令」的管理是缺乏的，「用户名+口令」的安全配置过程是缺失的。

Spring Boot也不提供对于我们自己开发的功能的任何防护功能。

一般来讲，一个安全的信道（信息传输的通道），需要通信双方在进行正式的信息传输之前对对方进行身份认证，服务提供方还需要在此基础之上，对请求方的请求进行权限的校验，以确保业务安全。这些内容也需要基于Spring Boot进行外围的安全扩展，例如采用前面提到的S-EDA进行进程级别的安全管控。这些还需要配套的安全服务提供支持。

一般来说，只要企业与互联网对接，那么随便一个面向消费者的「市场活动」，就有可能为企业带来井喷的流量。

传统企业内，更多的系统是管理信息类的支撑系统，这类系统在设计时的主要用户是企业内部员工以及有限的外部供应商。这类系统存在于企业内部的时间一直很长，功能耦合也很多，在功能解耦前，是非常不适合的，或者说绝对不可以直接为互联网的用户进行服务的。

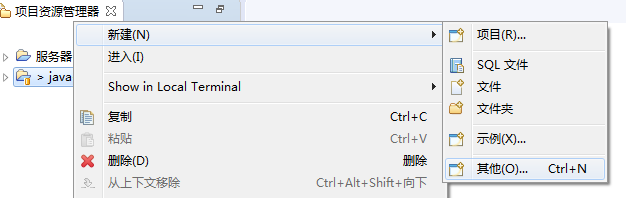
Spring Boot自身并没有提供这样的流控措施，所以需要结合前面提到的S-EDA进行流量的控制，并结合下层的水平扩展能力（例如，Kubernets）进行流量负载合理的动态扩容。

另外，在长业务流程的设计上，也尽可能地采用异步的方式，比如接口调用返回的是一个「受理号」，而不是业务的处理结果，避免井喷业务到来时，同步调用所带来的阻塞导致系统迅速崩溃，这些也都是Spring Boot自身并不解决的问题。

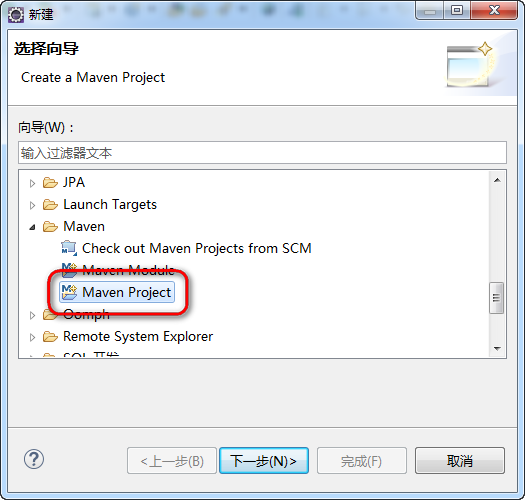
# 使用Eclipse开发Spring Boot工程

## 创建Maven工程

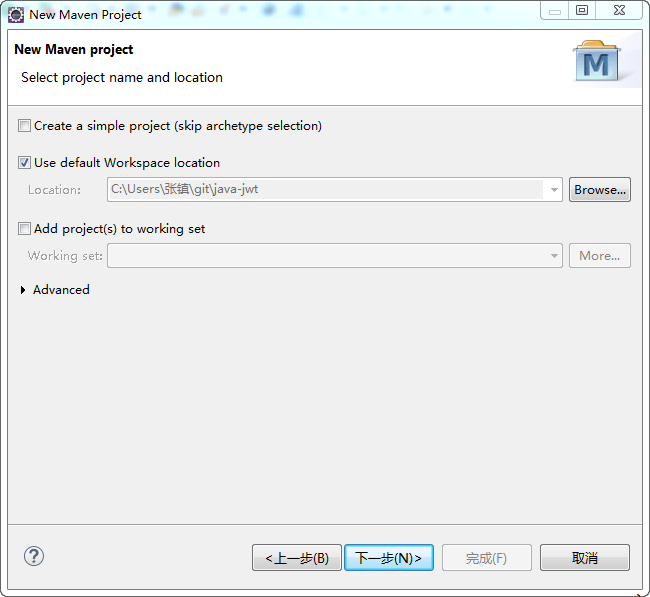
打开Eclipse，在“项目资源管理器”中，点击鼠标右键，选择“新建”-“其他”菜单。



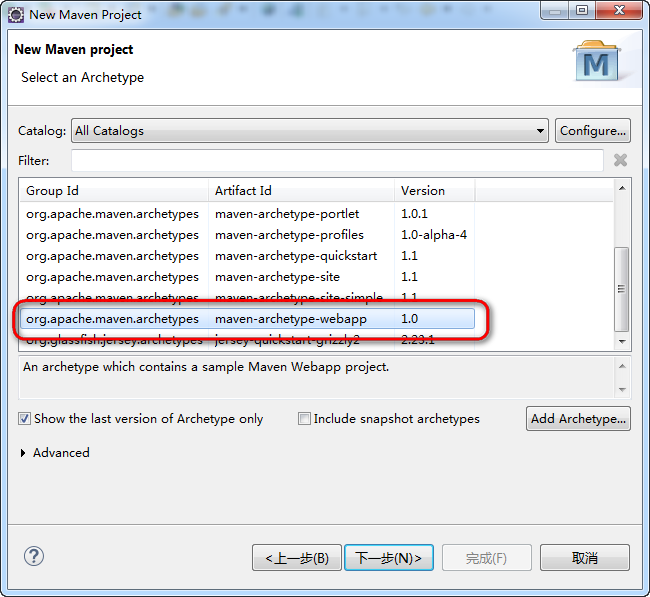
在弹出对话框中，选择“Maven”-“Maven Project”。



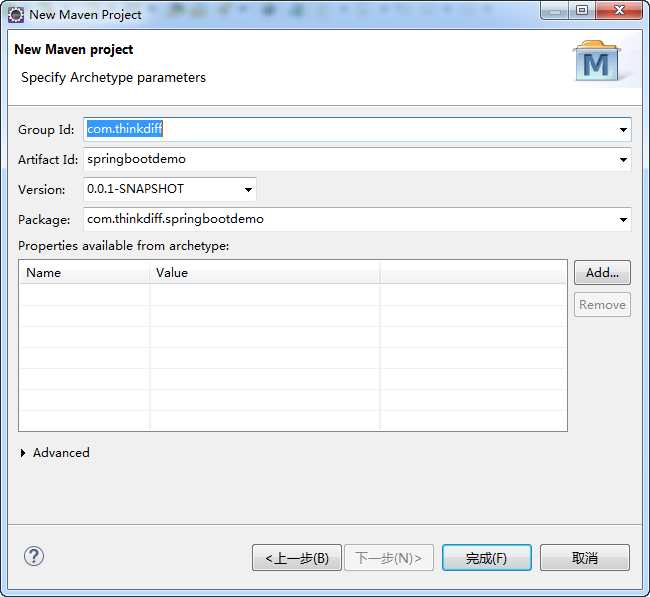
点击“下一步”按钮，出现下图，根据自己需要设置，可以使用默认的。



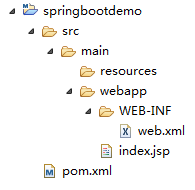
点击“下一步”按钮，在列表中选择“maven-archetype-webapp”项，创建一个web应用程序。



点击“下一步”按钮，设置项目的Group Id（一般输入公司跟URL）和Artifact Id（一般输入此项目的名称），其他项可以不用设置。系统自动将Group Id和Artifact Id组合成Package（包名）。

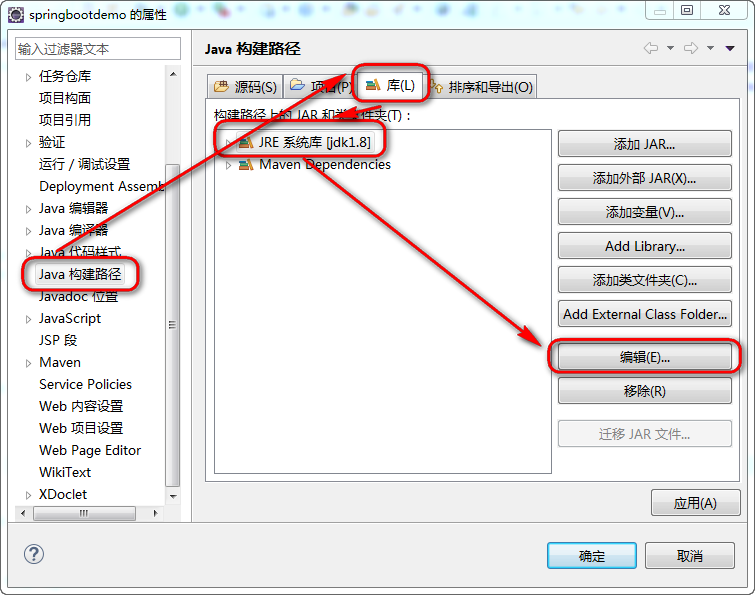


点击“完成”按钮，完成项目的创建。创建完毕的工程结构如下。

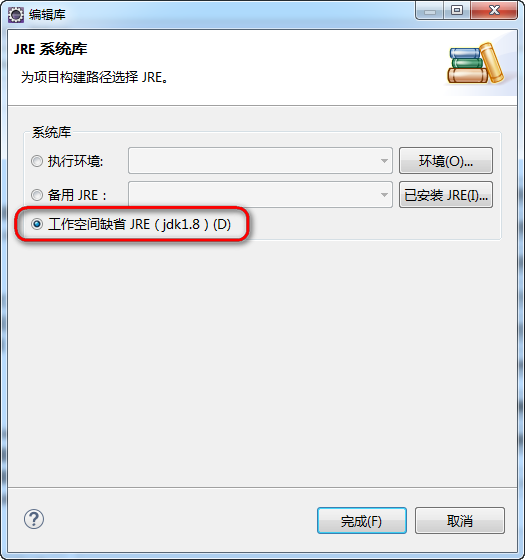


此时，我们发现，main文件夹下没有java子文件夹。这是一个Eclipse的Bug，系统默认选择了J2SE1.5，需要更改为现有的1.7或1.8。可以按照如下的操作方式解决。

在项目上右键选择“属性”（properties），然后点击“Java构建路径”（Java build path）。选中“库”（Librarys）选项页，选中“JRE系统库”（JRE System Library），点击“编辑”（Edit）按钮。



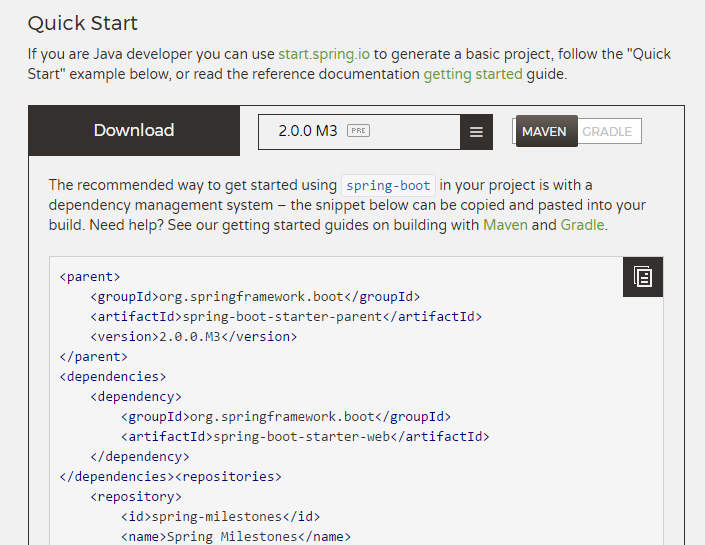
选择“工作空间缺省JRE（JDK1.8）”（workspace default jre），点击“完成”按钮就可以了。



## 修改pom.xml

访问<http://projects.spring.io/spring-boot/>，可以在此页面的“Quick Start”部分看到各个版本Spring Boot所需的Maven配置和示例Java代码。

本例使用最新的2.0.0.M3版。



在版本选择框中，选择“2.0.0 M3”，确保Download的方式选中“MAVEN”。此时，下方的配置文件框中，会显示Maven所需的配置内容。

系统自动生成的pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.thinkdiff</groupId>

<artifactId>springbootdemo</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>springbootdemo Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<finalName>springbootdemo</finalName>

</build>

</project>

将官网中获取的配置信息整合到现有的pom.xml中，形成如下内容：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.thinkdiff</groupId>

<artifactId>springbootdemo</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>springbootdemo Maven Webapp</name>

<url>http://maven.apache.org</url>

<!-- 父依赖 -->

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.0.M3</version>

</parent>

<dependencies>

<!-- Spring Boot启动 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/libs-milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<build>

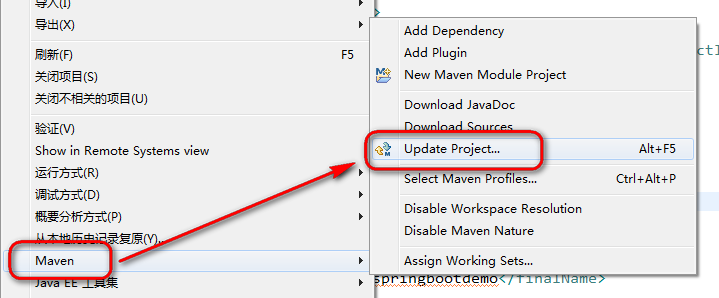
<finalName>springbootdemo</finalName>

</build>

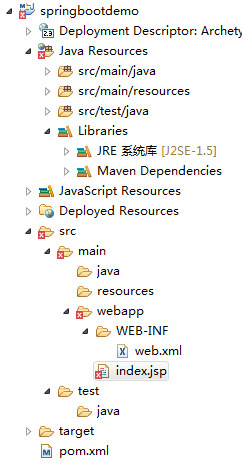
</project>

编写完后，保存pom.xml，系统即会自动下载所需jar包，自动构建工程。

如果没有自动构建，可以在项目点击邮件，选择Maven->Update Project，进行手动更新。



创建完毕，工程结构如下图：



工程中，系统自动创建的index.jsp没有实际用途，需要删除。

## 测试Spring Boot应用

在src/main/java目录下，新建一个com.thinkdiff.springbootdemo包，然后在包下面新建一个类SampleController。

代码如下：

**package** com.thinkdiff.springbootdemo;

**import** org.springframework.boot.\*;

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.stereotype.\*;

**import** org.springframework.web.bind.annotation.\*;

@Controller

@EnableAutoConfiguration

**public** **class** SampleController {

@RequestMapping("/")

@ResponseBody

String home() {

**return** "Hello World!";

}

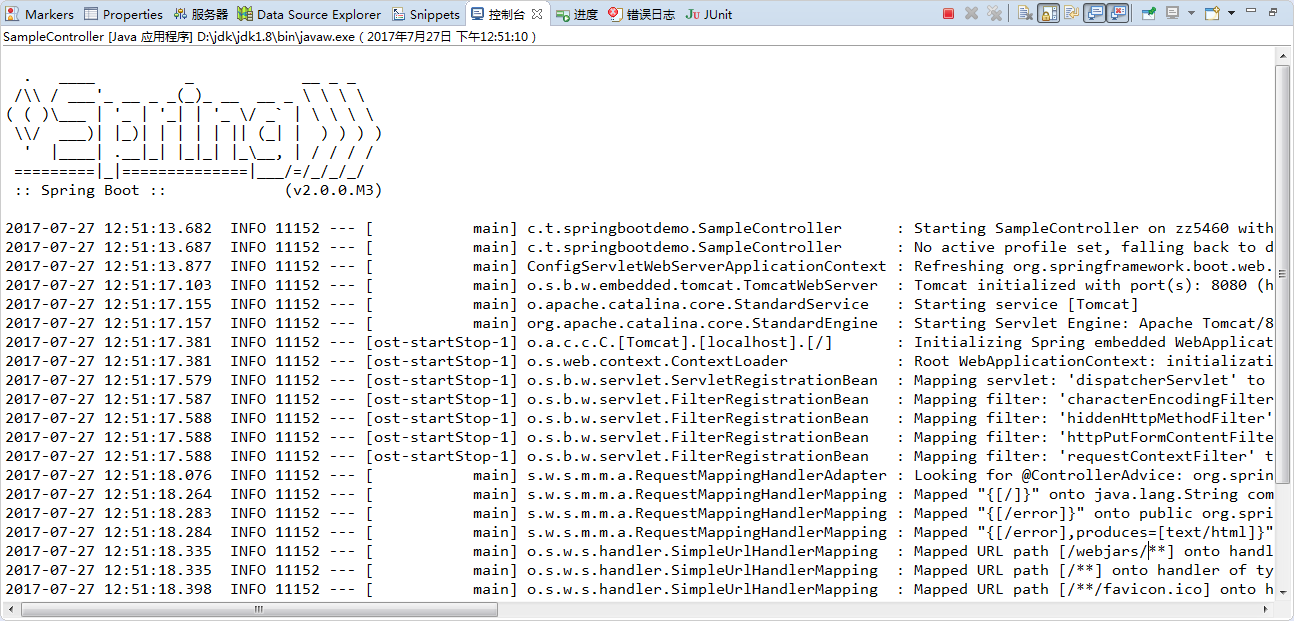
**public** **static** **void** main(String[] args) **throws** Exception {

SpringApplication.*run*(SampleController.**class**, args);

}

}

启动程序，右键AppleApplication.java文件，选择run as -> [Java](http://lib.csdn.net/base/java)Application。此时，控制台可以看到Spring Boot正在启动，并且输出如下形式内容：



如果没有报错的话，在浏览器中输入： <http://localhost:8080/> ，可以看到如下图的输出。



# Spring Boot使用

## Spring Boot初探

在Maven依赖中引入了spring-boot-starter-web，它包含了Spring Boot预定义的一些Web开发的常用依赖:

* spring-web, spring-webmvc Spring WebMvc框架
* tomcat-embed-\* 内嵌Tomcat容器
* jackson 处理json数据
* spring-\* Spring框架
* spring-boot-autoconfigure Spring Boot提供的自动配置功能

Java代码中没有任何配置，和传统的Spring应用相比，多了两个我们不认识的符号：

* @EnableAutoConfiguration
* SpringApplication

它们都是由Spring Boot框架提供。在SpringApplication.run()方法执行后，Spring Boot的autoconfigure发现这是一个Web应用（根据类路径上的依赖确定），于是在内嵌的Tomcat容器中启动了一个Spring的应用上下文，并且监听默认的tcp端口8080（默认约定）。同时在Spring Context中根据默认的约定配置了Spring WebMvc：

* Servlet容器默认的Context路径是/
* DispatherServlet匹配的路径(servlet-mapping中的url-patterns)是/\*
* @ComponentScan路径被默认设置为SampleController的同名package，也就是该package下的所有@Controller，@Service, @Component, @Repository都会被实例化后并加入Spring Context中。

没有一行配置代码、也没有web.xml。基于Spring Boot的应用在大多数情况下都不需要我们去显式地声明各类配置，而是**将最常用的默认配置作为约定**，在不声明的情况下也能适应大多数的开发场景。

## 数据库访问

除了最基本的Web框架，另一种非常普遍的开发场景是访问数据库。在传统的Spring应用中，访问数据库我们需要配置：

* 类路径上添加数据库访问驱动。
* 实例化DataSource对象，指定数据库url, username, password等信息。
* 注入JdbcTemplate对象，如果使用Hibernate，Mybatis等框架，还需要进一步配置框架信息。

spring-boot-starter-web-jdbc引入了spring-jdbc依赖，h2是一个内存关系型数据库。在引入了这些依赖并启动Spring Boot应用程序后，autoconfigure发现spring-jdbc位于类路径中，于是：

* 根据类路径上的JDBC驱动类型（预定义了derby, sqlite, mysql, oracle, sqlserver等等），创建一个DataSource连接池对象，如果是mysql, oracle等类型的数据库需要开发者配置相关信息。
* 在Spring Context中创建一个JdbcTemplate对象（使用DataSource初始化）

接下来开发者的工作就非常简单了，在业务逻辑中直接引入JdbcTemplate即可：

@Service

public class MyService {

@Autowired

JdbcTemplate jdbcTemplate;

}

除了spring-jdbc，Spring Boot还能够支持JPA，以及各种NoSQL数据库——包括MongoDB，Redis，全文索引工具elasticsearch, solr等等。

### 使用JPA

首先在Maven项目依赖中定义：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

在src/main/resources这个文件夹下面新建一个application.properties,并把相应的配置代码写进去。

#DB Configuration:

spring.datasource.driverClassName = com.mysql.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/testdb

spring.datasource.username = root

spring.datasource.password = 123456

#JPA Configuration:

spring.jpa.database=MySQL

spring.jpa.show-sql=true

spring.jpa.generate-ddl=true

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect

spring.jpa.hibernate.naming\_strategy=org.hibernate.cfg.ImprovedNamingStrategy

#spring.jpa.database=org.hibernate.dialect.MySQL5InnoDBDialect

#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MYSQL5Dialect

编写Model里面的UserLogin.java

package cn.springboot.SpringBootFirst.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

import javax.validation.constraints.NotNull;

@Entity

@Table(name="logintable")

public class UserLogin {

@Id

@NotNull

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

@NotNull

@Column(name="login\_name")

private String loginName;

@NotNull

@Column(name="login\_password")

private String loginPassword;

public UserLogin(){}

public UserLogin(String a, String b){

this.loginName = a;

this.loginPassword = b;

}

//Getter and Setter

}

编写DAO里面的 UserLoginDao.java

package cn.springboot.SpringBootFirst.dao;

import javax.transaction.Transactional;

import org.springframework.data.repository.CrudRepository;

import cn.springboot.SpringBootFirst.model.UserLogin;

@Transactional

public interface UserLoginDao extends CrudRepository<UserLogin, Long>{

UserLogin findByloginName(String loginName);

}

通常我们会在持久层声明一个接口（如上面代码中的interface UserLoginDao),然后让该接口来继承接口Repository（此接口是Spring Data JPA）中最为核心的接口，但是它只是一个标记型接口，其中包含任何方法，当然如果有需要，Spring Data也提供了很多它的子接口，其中封装了常用的增删改查和分页相关的方法。

正如上面我的代码中所写的那样，我实现的是CrudRepository接口。   
如果持久层接口较多，且每一个接口都需要声明相似的增删改查方法，直接继承 Repository 就显得有些啰嗦，这时可以继承CrudRepository，它会自动为域对象创建增删改查方法，供业务层直接使用。开发者只是多写了 “Crud” 四个字母，即刻便为域对象提供了开箱即用的十个增删改查方法。

但是，使用 CrudRepository也有副作用，它可能暴露了你不希望暴露给业务层的方法。比如某些接口你只希望提供增加的操作而不希望提供删除的方法。针对这种情况，开发者只能退回到 Repository 接口，然后到 CrudRepository 中把希望保留的方法声明复制到自定义的接口中即可。分页查询和排序是持久层常用的功能，Spring Data 为此提供了 PagingAndSortingRepository 接口，它继承自 CrudRepository 接口，在 CrudRepository 基础上新增了两个与分页有关的方法。但是，我们很少会将自定义的持久层接口直接继承自 PagingAndSortingRepository，而是在继承 Repository 或 CrudRepository 的基础上，在自己声明的方法参数列表最后增加一个 Pageable 或 Sort 类型的参数，用于指定分页或排序信息即可，这比直接使用 PagingAndSortingRepository 提供了更大的灵活性。

JpaRepository 是继承自 PagingAndSortingRepository 的针对 JPA 技术提供的接口，它在父接口的基础上，提供了其他一些方法，比如 flush()，saveAndFlush()，deleteInBatch() 等。如果有这样的需求，则可以继承该接口。

上述四个接口，开发者到底该如何选择？其实依据很简单，根据具体的业务需求，选择其中之一。笔者建议在通常情况下优先选择 Repository 接口。因为 Repository 接口已经能满足日常需求，其他接口能做到的在 Repository 中也能做到，彼此之间并不存在功能强弱的问题。只是 Repository 需要显示声明需要的方法，而其他则可能已经提供了相关的方法，不需要再显式声明，但如果对 Spring Data JPA 不熟悉，别人在检视代码或者接手相关代码时会有疑惑，他们不明白为什么明明在持久层接口中声明了三个方法，而在业务层使用该接口时，却发现有七八个方法可用，从这个角度而言，应该优先考虑使用 Repository 接口。

编写Controller中的UserLoginController.java

package cn.springboot.SpringBootFirst.controller;

import java.util.Iterator;

import javax.annotation.Resource;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import cn.springboot.SpringBootFirst.model.UserLogin;

import cn.springboot.SpringBootFirst.dao.UserLoginDao;

@RestController

public class UserLoginController {

@Resource

UserLoginDao userLoginDAO;

@RequestMapping("/userLogin")

@ResponseBody

public String login(String userName, String password){

UserLogin ul = userLoginDAO.findByloginName(userName);

/\*String result = "";

Iterable<UserLogin> it = userLoginDAO.findAll();

for(UserLogin ul:it){

result += ul.getLogin\_name();

}\*/

if(ul==null)

{

return "Login Error";

}

else

{

return ul.getLoginName()+" "+ul.getLoginPassword();

}

//return result;

}

}

最后在浏览器中输入：<http://localhost:8080/userLogin?userName=wang&password=123456>

即可测试功能是否正常。

### 使用JdbcTemplate

首先在Maven项目依赖中定义：

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

添加配置文件配置数据库和其他参数。在resource文件夹下添加application.properties配置文件并输入数据库参数，如下：

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test

spring.datasource.username=root

spring.datasource.password=123456

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.max-idle=10

spring.datasource.max-wait=10000

spring.datasource.min-idle=5

spring.datasource.initial-size=5

server.port=8011

server.session.timeout=10

server.tomcat.uri-encoding=UTF-8

新建Controller类测试数据库连接

package com.lgp.SpringBoot;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/mydb")

public class DbController {

@Autowired

private JdbcTemplate jdbcTemplate;

@RequestMapping("/getUsers")

public List<Map<String, Object>> getDbType(){

String sql = "select \* from appuser";

List<Map<String, Object>> list = jdbcTemplate.queryForList(sql);

for (Map<String, Object> map : list) {

Set<Entry<String, Object>> entries = map.entrySet( );

if(entries != null) {

Iterator<Entry<String, Object>> iterator = entries.iterator( );

while(iterator.hasNext( )) {

Entry<String, Object> entry =(Entry<String, Object>) iterator.next( );

Object key = entry.getKey( );

Object value = entry.getValue();

System.out.println(key+":"+value);

}

}

}

return list;

}

@RequestMapping("/user/{id}")

public Map<String,Object> getUser(@PathVariable String id){

Map<String,Object> map = null;

List<Map<String, Object>> list = getDbType();

for (Map<String, Object> dbmap : list) {

Set<String> set = dbmap.keySet();

for (String key : set) {

if(key.equals("id")){

if(dbmap.get(key).equals(id)){

map = dbmap;

}

}

}

}

if(map==null)

map = list.get(0);

return map;

}

}

运行App 输入地址 输出数据库数据。

### 使用Mybatis

首先在Maven项目依赖中定义（请注意版本号，建议使用新版）：

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.0.0</version>

</dependency>

在配置文件中添加配置信息：

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test

spring.datasource.username=root

spring.datasource.password=123456

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.max-idle=10

spring.datasource.max-wait=10000

spring.datasource.min-idle=5

spring.datasource.initial-size=5

server.port=8011

server.session.timeout=10

server.tomcat.uri-encoding=UTF-8

# mybatis.config= classpath:mybatis-config.xml

mybatis.mapperLocations=classpath:mappers/\*.xml

# domain object's package

mybatis.typeAliasesPackage=com.lgp.SpringBoot.bean

# handler's package

# mybatis.typeHandlersPackage=

# check the mybatis configuration exists

# mybatis.check-config-location=

# mode of execution. Default is SIMPLE

# mybatis.executorType=

依次添加mapper的接口类和xml文件

**AppMessageMapper.java**

package com.lgp.SpringBoot.mapper;

import java.util.List;

import com.lgp.SpringBoot.bean.AppMessage;

public interface AppMessageMapper {

int deleteByPrimaryKey(String id);

int insert(AppMessage record);

int insertSelective(AppMessage record);

AppMessage selectByPrimaryKey(String id);

int updateByPrimaryKeySelective(AppMessage record);

int updateByPrimaryKey(AppMessage record);

List<AppMessage> selectAll();

List<AppMessage> getMessById(String id);

}

**AppMessageMapper.xml**

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="com.lgp.SpringBoot.mapper.AppMessageMapper" >

<resultMap id="BaseResultMap" type="com.lgp.SpringBoot.bean.AppMessage" >

<id column="id" property="id" jdbcType="VARCHAR" />

<result column="message" property="message" jdbcType="VARCHAR" />

<result column="senddate" property="senddate" jdbcType="TIMESTAMP" />

</resultMap>

<sql id="Base\_Column\_List" >

id, message, senddate

</sql>

<select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.String" >

select

<include refid="Base\_Column\_List" />

from appuser\_message

where id = #{id,jdbcType=VARCHAR}

</select>

<delete id="deleteByPrimaryKey" parameterType="java.lang.String" >

delete from appuser\_message

where id = #{id,jdbcType=VARCHAR}

</delete>

<insert id="insert" parameterType="com.lgp.SpringBoot.bean.AppMessage" >

insert into appuser\_message (id, message, senddate

)

values (#{id,jdbcType=VARCHAR}, #{message,jdbcType=VARCHAR}, #{senddate,jdbcType=TIMESTAMP}

)

</insert>

<insert id="insertSelective" parameterType="com.lgp.SpringBoot.bean.AppMessage" >

insert into appuser\_message

<trim prefix="(" suffix=")" suffixOverrides="," >

<if test="id != null" >

id,

</if>

<if test="message != null" >

message,

</if>

<if test="senddate != null" >

senddate,

</if>

</trim>

<trim prefix="values (" suffix=")" suffixOverrides="," >

<if test="id != null" >

#{id,jdbcType=VARCHAR},

</if>

<if test="message != null" >

#{message,jdbcType=VARCHAR},

</if>

<if test="senddate != null" >

#{senddate,jdbcType=TIMESTAMP},

</if>

</trim>

</insert>

<update id="updateByPrimaryKeySelective" parameterType="com.lgp.SpringBoot.bean.AppMessage" >

update appuser\_message

<set >

<if test="message != null" >

message = #{message,jdbcType=VARCHAR},

</if>

<if test="senddate != null" >

senddate = #{senddate,jdbcType=TIMESTAMP},

</if>

</set>

where id = #{id,jdbcType=VARCHAR}

</update>

<update id="updateByPrimaryKey" parameterType="com.lgp.SpringBoot.bean.AppMessage" >

update appuser\_message

set message = #{message,jdbcType=VARCHAR},

senddate = #{senddate,jdbcType=TIMESTAMP}

where id = #{id,jdbcType=VARCHAR}

</update>

<select id="selectAll" resultMap="BaseResultMap">

select

id, message, senddate

from appuser\_message

order by senddate asc

</select>

<select id="getMessById" resultMap="BaseResultMap" parameterType="java.lang.String">

select

id, message, senddate

from

appuser\_message

where id = #{id,jdbcType=VARCHAR}

order by senddate asc

</select>

</mapper>

**AppMessage.java**

package com.lgp.SpringBoot.bean;

import java.util.Date;

public class AppMessage {

private String id;

private String message;

private Date senddate;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id == null ? null : id.trim();

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message == null ? null : message.trim();

}

public Date getSenddate() {

return senddate;

}

public void setSenddate(Date senddate) {

this.senddate = senddate;

}

}

**AppMessageService.java**

package com.lgp.SpringBoot.service;

import java.util.ArrayList;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.lgp.SpringBoot.bean.AppMessage;

import com.lgp.SpringBoot.mapper.AppMessageMapper;

@Service

public class AppMessageService {

@Autowired

private AppMessageMapper mapper;

public List<AppMessage> getMessage(){

List<AppMessage> list = new ArrayList<AppMessage>();

list.add(mapper.selectByPrimaryKey("xtt"));

//list = mapper.selectAll();

return list;

}

public List<AppMessage> getAllMessage(){

List<AppMessage> list = new ArrayList<AppMessage>();

list = mapper.selectAll();

return list;

}

public int addMessage(AppMessage appMessage) {

return mapper.insert(appMessage);

}

public List<AppMessage> getMessageById(String id) {

return mapper.getMessById(id);

}

public int delMessage(String id) {

return mapper.deleteByPrimaryKey(id);

}

}

**APPMessageController.java**

package com.lgp.SpringBoot.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import com.lgp.SpringBoot.bean.AppMessage;

import com.lgp.SpringBoot.service.AppMessageService;

@RestController

@RequestMapping("/appmessage")

public class APPMessageController {

@Autowired

private AppMessageService service;

@RequestMapping("/getThree")

public List<AppMessage> getThreeForMessage(){

List<AppMessage> list = service.getMessage();

return list;

}

@RequestMapping("/getAll")

public List<AppMessage> getAllMessage(){

List<AppMessage> list = service.getAllMessage();

int num = list.size();

if(null!=list && num>3){

for (int i = 0; i < num-3; i++) {

list.remove(0);

}

}

return list;

}

@RequestMapping("/getByID")

public List<AppMessage> getMessageById(@RequestParam("id") String id){

List<AppMessage> list = service.getMessageById(id);

int num = list.size();

if(null!=list && num>5){

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

list.remove(0);

}

}

return list;

}

@RequestMapping(value = "/add",method = RequestMethod.POST)

public int addMessage(@RequestBody AppMessage appMessage){

return service.addMessage(appMessage);

}

@RequestMapping(value="/delMessageById",method=RequestMethod.POST)

public int delMessageById(@RequestParam("id") String id){

return service.delMessage(id);

}

}

## 配置

Spring Boot最大的特色是“约定优先配置”，大量的默认配置对开发者十分的友好。但是在实际的应用开发过程中，默认配置不可能满足所有场景，同时用户也需要配置一些必须的配置项——例如数据库连接信息。Spring Boot的配置系统能够让开发者快速的覆盖默认约定，同时支持Properties配置文件和YAML配置文件两种格式，默认情况下Spring Boot加载类路径上的application.properties或application.yml文件，例如：

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

YAML格式更加简洁：

spring:

datasource:

url: jdbc:mysql://localhost/test

username: dbuser

password: dbpass

driver-class: com.mysql.jdbc.Driver

一旦发现这些信息，Spring Boot就会根据它们创建DataSource对象。另一个常见的配置场景是Web应用服务器：

# Server settings (ServerProperties)

server:

port: 8080

address: 127.0.0.1

sessionTimeout: 30

contextPath: /

# Tomcat specifics

tomcat:

accessLogEnabled: false

protocolHeader: x-forwarded-proto

remoteIpHeader: x-forwarded-for

basedir:

backgroundProcessorDelay: 30 # secs

通过port和address可以修改服务器监听的地址和端口，sessionTimeout配置session过期时间（再也不用修改web.xml了，因为它根本不存在）。同时如果在生产环境中使用内嵌Tomcat，当然希望能够配置它的日志、线程池等信息，这些现在都可以通过Spring Boot的属性文件配置，而不再需要再对生产环境中的Tomcat实例进行单独的配置管理了。

## @EnableAutoCongiguration

从Spring 3.0开始，为了替代繁琐的XML配置，引入了@Enable...注解对@Configuration类进行修饰以达到和XML配置相同的效果。想必不少开发者已经使用过类似注解：

* @EnableTransactionManagement开启Spring事务管理，相当于XMl中的<tx:\*>
* @EnableWebMvc使用Spring MVC框架的一些默认配置
* @EnableScheduling会初始化一个Scheduler用于执行定时任务和异步任务

Spring Boot提供的@EnableAutoCongiguration似乎功能更加强大，一旦加上，上述所有的配置似乎都被包含进来而无需开发者显式声明。它究竟是如何做到的呢，先看看它的定义：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Import({ EnableAutoConfigurationImportSelector.class,

AutoConfigurationPackages.Registrar.class })

public @interface EnableAutoConfiguration {

/\*\*

\* Exclude specific auto-configuration classes such that they will never be applied.

\*/

Class<?>[] exclude() default {};

}

EnableAutoConfigurationImportSelector使用的是spring-core模块中的SpringFactoriesLoader#loadFactoryNames()方法，它的作用是在类路径上扫描META-INF/spring.factories文件中定义的类：

# Initializers

org.springframework.context.ApplicationContextInitializer=\

org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

org.springframework.boot.autoconfigure.data.JpaRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.data.MongoRepositoriesAutoConfiguration,\

org.springframework.boot.autoconfigure.redis.RedisAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\

org.springframework.boot.autoconfigure.jms.JmsTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\

org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\

org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\

org.springframework.boot.autoconfigure.mongo.MongoTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\

org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\

org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\

org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\

org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration

实际上这就是Spring Boot会自动配置的一些对象，例如前面提到的Web框架由

EmbeddedServletContainerAutoConfiguration,

DispatcherServletAutoConfiguration,

ServerPropertiesAutoConfiguration

等配置完成，而DataSource的自动配置则是由DataSourceAutoConfiguration完成。现在我们以Mongo的配置MongoAutoConfiguration为例，来探索Spring Boot是如何完成这些配置的：

@Configuration

@ConditionalOnClass(Mongo.class)

@EnableConfigurationProperties(MongoProperties.class)

public class MongoAutoConfiguration {

@Autowired

private MongoProperties properties;

private Mongo mongo;

@PreDestroy

public void close() throws UnknownHostException {

if (this.mongo != null) {

this.mongo.close();

}

}

@Bean

@ConditionalOnMissingBean

public Mongo mongo() throws UnknownHostException {

this.mongo = this.properties.createMongoClient();

return this.mongo;

}

}

首先这是一个Spring的配置@Configuration，它定义了我们访问Mongo需要的@Bean，如果这个@Configuration被Spring Context扫描到，那么Context中自然也就有两个一个Mongo对象能够直接为开发者所用。

但是注意到其它几个Spring注解：

* @ConditionOnClass表明该@Configuration仅仅在一定条件下才会被加载，这里的条件是Mongo.class位于类路径上
* @EnableConfigurationProperties将Spring Boot的配置文件（application.properties）中的spring.data.mongodb.\*属性映射为MongoProperties并注入到MongoAutoConfiguration中。
* @ConditionalOnMissingBean说明Spring Boot仅仅在当前上下文中不存在Mongo对象时，才会实例化一个Bean。这个逻辑也体现了Spring Boot的另外一个特性——自定义的Bean优先于框架的默认配置，我们如果显式的在业务代码中定义了一个Mongo对象，那么Spring Boot就不再创建。

接下来看一看MongoProperties：

@ConfigurationProperties(prefix = "spring.data.mongodb")

public class MongoProperties {

private String host;

private int port = DBPort.PORT;

private String uri = "mongodb://localhost/test";

private String database;

// ... getters/ setters omitted

}

显然，它就是以spring.data.mongodb作为前缀的属性，然后通过名字直接映射为对象的属性，同时还包含了一些默认值。如果不配置，那么mongo.uri就是mongodb://localhost/test。

## Production特性

从前面的例子可以看出，Spring Boot能够非常快速的做出一些原型应用，但是它同样可以被用于生产环境。为了添加生产环境特性支持，需要在Maven依赖中引入：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

加入actuator依赖后，应用启动后会创建一些基于Web的Endpoint：

* /autoconfig，用来查看Spring Boot的框架自动配置信息，哪些被自动配置，哪些没有，原因是什么。
* /beans，显示应用上下文的Bean列表
* /dump，显示线程dump信息
* /health，应用健康状况检查
* /metrics
* /shutdown, 默认没有打开
* /trace

## 通过Actuator对系统进行监控

Spring Boot有四大神器，分别是auto-configuration、starters、cli、actuator，本文主要讲actuator。actuator是spring boot提供的对应用系统的自省和监控的集成功能，可以对应用系统进行配置查看、相关功能统计等。

### 使用actuator

添加依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

### 主要暴露的功能

| **HTTP方法** | **路径** | **描述** | **鉴权** |
| --- | --- | --- | --- |
| GET | /autoconfig | 查看自动配置的使用情况 | true |
| GET | /configprops | 查看配置属性，包括默认配置 | true |
| GET | /beans | 查看bean及其关系列表 | true |
| GET | /dump | 打印线程栈 | true |
| GET | /env | 查看所有环境变量 | true |
| GET | /env/{name} | 查看具体变量值 | true |
| GET | /health | 查看应用健康指标 | false |
| GET | /info | 查看应用信息 | false |
| GET | /mappings | 查看所有url映射 | true |
| GET | /metrics | 查看应用基本指标 | true |
| GET | /metrics/{name} | 查看具体指标 | true |
| POST | /shutdown | 关闭应用 | true |
| GET | /trace | 查看基本追踪信息 | true |

#### 返回数据示例：/autoconfig

{

"positiveMatches": {

"AuditAutoConfiguration.AuditEventRepositoryConfiguration": [

{

"condition": "OnBeanCondition",

"message": "@ConditionalOnMissingBean (types: org.springframework.boot.actuate.audit.AuditEventRepository; SearchStrategy: all) found no beans"

}

]

},

"negativeMatches": {

"CacheStatisticsAutoConfiguration": [

{

"condition": "OnBeanCondition",

"message": "@ConditionalOnBean (types: org.springframework.cache.CacheManager; SearchStrategy: all) found no beans"

}

]

}

}

#### 返回数据示例：/configprops

{

"management.health.status.CONFIGURATION\_PROPERTIES": {

"prefix": "management.health.status",

"properties": {

"order": null

}

},

"multipart.CONFIGURATION\_PROPERTIES": {

"prefix": "multipart",

"properties": {

"enabled": false,

"maxRequestSize": "10Mb",

"location": null,

"fileSizeThreshold": "0",

"maxFileSize": "1Mb"

}

},

"environmentEndpoint": {

"prefix": "endpoints.env",

"properties": {

"id": "env",

"enabled": true,

"sensitive": true

}

}

}

#### 返回数据示例：/beans

[

{

"context": "application:8080",

"parent": null,

"beans": [

{

"bean": "appMain",

"scope": "singleton",

"type": "com.xixicat.AppMain$$EnhancerBySpringCGLIB$$29382b14",

"resource": "null",

"dependencies": [ ]

},

{

"bean": "videoInfoMapper",

"scope": "singleton",

"type": "com.xixicat.dao.VideoInfoMapper",

"resource": "file [/Users/xixicat/workspace/video-uber/target/classes/com/xixicat/dao/VideoInfoMapper.class]",

"dependencies": [

"sqlSessionFactory"

]

}

]

}

]

#### 返回数据示例：/dump

[

{

"threadName": "Signal Dispatcher",

"threadId": 4,

"blockedTime": -1,

"blockedCount": 0,

"waitedTime": -1,

"waitedCount": 0,

"lockName": null,

"lockOwnerId": -1,

"lockOwnerName": null,

"inNative": false,

"suspended": false,

"threadState": "RUNNABLE",

"stackTrace": [ ],

"lockedMonitors": [ ],

"lockedSynchronizers": [ ],

"lockInfo": null

},

{

"threadName": "Reference Handler",

"threadId": 2,

"blockedTime": -1,

"blockedCount": 217,

"waitedTime": -1,

"waitedCount": 9,

"lockName": "java.lang.ref.Reference$Lock@45de945",

"lockOwnerId": -1,

"lockOwnerName": null,

"inNative": false,

"suspended": false,

"threadState": "WAITING",

"stackTrace": [

{

"methodName": "wait",

"fileName": "Object.java",

"lineNumber": -2,

"className": "java.lang.Object",

"nativeMethod": true

},

{

"methodName": "wait",

"fileName": "Object.java",

"lineNumber": 503,

"className": "java.lang.Object",

"nativeMethod": false

},

{

"methodName": "run",

"fileName": "Reference.java",

"lineNumber": 133,

"className": "java.lang.ref.Reference$ReferenceHandler",

"nativeMethod": false

}

],

"lockedMonitors": [ ],

"lockedSynchronizers": [ ],

"lockInfo": {

"className": "java.lang.ref.Reference$Lock",

"identityHashCode": 73263429

}

}

]

#### 返回数据示例：/env

{

profiles: [],

server.ports: {

local.server.port: 8080

},

servletContextInitParams: {},

systemProperties: {

java.runtime.name: "Java(TM) SE Runtime Environment",

sun.boot.library.path: "/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib",

java.vm.version: "24.79-b02",

gopherProxySet: "false",

maven.multiModuleProjectDirectory: "/Users/xixicat/workspace/video-uber",

java.vm.vendor: "Oracle Corporation",

java.vendor.url: "http://java.oracle.com/",

guice.disable.misplaced.annotation.check: "true",

path.separator: ":",

java.vm.name: "Java HotSpot(TM) 64-Bit Server VM",

file.encoding.pkg: "sun.io",

user.country: "CN",

sun.java.launcher: "SUN\_STANDARD",

sun.os.patch.level: "unknown",

PID: "763",

java.vm.specification.name: "Java Virtual Machine Specification",

user.dir: "/Users/xixicat/workspace/video-uber",

java.runtime.version: "1.7.0\_79-b15",

java.awt.graphicsenv: "sun.awt.CGraphicsEnvironment",

java.endorsed.dirs: "/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/endorsed",

os.arch: "x86\_64",

java.io.tmpdir: "/var/folders/tl/xkf4nr61033gd6lk5d3llz080000gn/T/",

line.separator: " ",

java.vm.specification.vendor: "Oracle Corporation",

os.name: "Mac OS X",

classworlds.conf: "/Users/xixicat/devtool/maven-3.3.3/bin/m2.conf",

sun.jnu.encoding: "UTF-8",

spring.beaninfo.ignore: "true",

java.library.path: "/Users/xixicat/Library/Java/Extensions:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java:.",

java.specification.name: "Java Platform API Specification",

java.class.version: "51.0",

sun.management.compiler: "HotSpot 64-Bit Tiered Compilers",

os.version: "10.10.5",

user.home: "/Users/xixicat",

user.timezone: "Asia/Shanghai",

java.awt.printerjob: "sun.lwawt.macosx.CPrinterJob",

file.encoding: "UTF-8",

java.specification.version: "1.7",

java.class.path: "/Users/xixicat/devtool/maven-3.3.3/boot/plexus-classworlds-2.5.2.jar",

user.name: "xixicat",

java.vm.specification.version: "1.7",

sun.java.command: "org.codehaus.plexus.classworlds.launcher.Launcher spring-boot:run",

java.home: "/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre",

sun.arch.data.model: "64",

user.language: "zh",

java.specification.vendor: "Oracle Corporation",

awt.toolkit: "sun.lwawt.macosx.LWCToolkit",

java.vm.info: "mixed mode",

java.version: "1.7.0\_79",

java.ext.dirs: "/Users/xixicat/Library/Java/Extensions:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/ext:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java",

sun.boot.class.path: "/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/resources.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/rt.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/sunrsasign.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/jsse.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/jce.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/charsets.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/lib/jfr.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home/jre/classes",

java.awt.headless: "true",

java.vendor: "Oracle Corporation",

maven.home: "/Users/xixicat/devtool/maven-3.3.3",

file.separator: "/",

LOG\_EXCEPTION\_CONVERSION\_WORD: "%wEx",

java.vendor.url.bug: "http://bugreport.sun.com/bugreport/",

sun.io.unicode.encoding: "UnicodeBig",

sun.cpu.endian: "little",

sun.cpu.isalist: ""

},

systemEnvironment: {

TERM: "xterm-256color",

ZSH: "/Users/xixicat/.oh-my-zsh",

GVM\_BROKER\_SERVICE: "http://release.gvm.io",

GRIFFON\_HOME: "/Users/xixicat/.gvm/griffon/current",

JAVA\_MAIN\_CLASS\_763: "org.codehaus.plexus.classworlds.launcher.Launcher",

JAVA\_HOME: "/Library/Java/JavaVirtualMachines/jdk1.7.0\_79.jdk/Contents/Home",

SHLVL: "1",

MAVEN\_CMD\_LINE\_ARGS: " spring-boot:run",

\_\_CF\_USER\_TEXT\_ENCODING: "0x1F5:0x19:0x34",

GROOVY\_HOME: "/Users/xixicat/.gvm/groovy/current",

XPC\_FLAGS: "0x0",

GVM\_INIT: "true",

JBAKE\_HOME: "/Users/xixicat/.gvm/jbake/current",

PWD: "/Users/xixicat/workspace/video-uber",

GVM\_DIR: "/Users/xixicat/.gvm",

GVM\_VERSION: "2.4.3",

MAVEN\_PROJECTBASEDIR: "/Users/xixicat/workspace/video-uber",

LOGNAME: "xixicat",

SSH\_AUTH\_SOCK: "/private/tmp/com.apple.launchd.93xr1duECQ/Listeners",

SPRINGBOOT\_HOME: "/Users/xixicat/.gvm/springboot/current",

GAIDEN\_HOME: "/Users/xixicat/.gvm/gaiden/current",

LAZYBONES\_HOME: "/Users/xixicat/.gvm/lazybones/current",

OLDPWD: "/Users/xixicat/workspace/video-uber",

SHELL: "/bin/zsh",

JBOSSFORGE\_HOME: "/Users/xixicat/.gvm/jbossforge/current",

LC\_CTYPE: "zh\_CN.UTF-8",

TMPDIR: "/var/folders/tl/xkf4nr61033gd6lk5d3llz080000gn/T/",

GVM\_SERVICE: "http://api.gvmtool.net",

GVM\_PLATFORM: "Darwin",

CLASSPATH: ".:/Users/xixicat/.m2/repository/co/paralleluniverse/quasar-core/0.7.2/quasar-core-0.7.2.jar",

GLIDE\_HOME: "/Users/xixicat/.gvm/glide/current",

PATH: "/Users/xixicat/.gvm/vertx/current/bin:/Users/xixicat/.gvm/springboot/current/bin:/Users/xixicat/.gvm/lazybones/current/bin:/Users/xixicat/.gvm/jbossforge/current/bin:/Users/xixicat/.gvm/jbake/current/bin:/Users/xixicat/.gvm/groovyserv/current/bin:/Users/xixicat/.gvm/groovy/current/bin:/Users/xixicat/.gvm/griffon/current/bin:/Users/xixicat/.gvm/grails/current/bin:/Users/xixicat/.gvm/gradle/current/bin:/Users/xixicat/.gvm/glide/current/bin:/Users/xixicat/.gvm/gaiden/current/bin:/Users/xixicat/.gvm/crash/current/bin:/Users/xixicat/.gvm/asciidoctorj/current/bin:/Users/xixicat/bin:/usr/local/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/git/bin:/Users/xixicat/devtool/maven-3.3.3/bin:/Users/xixicat/devtool/gradle-2.6/bin:/Users/xixicat/devtool/android-sdk/platform-tools",

GRADLE\_HOME: "/Users/xixicat/.gvm/gradle/current",

GROOVYSERV\_HOME: "/Users/xixicat/.gvm/groovyserv/current",

GRAILS\_HOME: "/Users/xixicat/.gvm/grails/current",

USER: "xixicat",

LESS: "-R",

PAGER: "less",

HOME: "/Users/xixicat",

CRASH\_HOME: "/Users/xixicat/.gvm/crash/current",

XPC\_SERVICE\_NAME: "0",

VERTX\_HOME: "/Users/xixicat/.gvm/vertx/current",

GVM\_BROADCAST\_SERVICE: "http://cast.gvm.io",

Apple\_PubSub\_Socket\_Render: "/private/tmp/com.apple.launchd.y6fNwP8Sk6/Render",

LSCOLORS: "Gxfxcxdxbxegedabagacad",

ASCIIDOCTORJ\_HOME: "/Users/xixicat/.gvm/asciidoctorj/current"

},

applicationConfig: [classpath: /application.properties]: {

pool.acquireIncrement: "1",

pool.minPoolSize: "5",

pool.initialPoolSize: "1",

database.username: "root",

pool.maxIdleTime: "60",

database.url: "jdbc:mysql://127.0.0.1:3307/video\_uber?autoReconnect=true&useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull",

spring.jackson.dateFormat: "yyyy-MM-dd'T'HH:mm:ss",

database.slave.username: "root",

spring.jackson.serialization.write - dates - as - timestamps: "false",

pool.idleTimeout: "30000",

database.slave.url: "jdbc:mysql://127.0.0.1:3307/demo?autoReconnect=true&useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull",

server.port: "8080",

database.slave.password: "\*\*\*\*\*\*",

database.password: "\*\*\*\*\*\*",

database.driverClassName: "com.mysql.jdbc.Driver",

pool.maxPoolSize: "50",

database.dataSourceClassName: "com.mysql.jdbc.jdbc2.optional.MysqlDataSource"

}

}

#### 返回数据示例：/health

{

status: "UP",

diskSpace: {

status: "UP",

total: 249779191808,

free: 193741590528,

threshold: 10485760

},

db: {

status: "UP",

database: "MySQL",

hello: 1

}

}

#### 返回数据示例：/info

需要自己在application.properties里头添加信息，比如

info:

contact:

email: xixicat@gmail.com

phone: 0755-82890987

然后请求就可以出来了

{

"contact": {

"phone": "0755-82890987",

"email": "xixicat@gmail.com"

}

}

#### 返回数据示例：/mappings

{

{

[/metrics || /metrics.json], methods = [GET], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"

}, {

[/beans || /beans.json], methods = [GET], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"

}, {

[/health || /health.json], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.HealthMvcEndpoint.invoke(java.security.Principal)"

}, {

[/info || /info.json], methods = [GET], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"

}, {

[/trace || /trace.json], methods = [GET], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"

}, {

[/autoconfig || /autoconfig.json], methods = [GET], produces = [application / json]

}: {

bean: "endpointHandlerMapping",

method: "public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"

}

}

#### 返回数据示例：/metrics

{

mem: 499404,

mem.free: 257591,

processors: 8,

instance.uptime: 4284997,

uptime: 4294909,

systemload.average: 1.84521484375,

heap.committed: 437248,

heap.init: 262144,

heap.used: 179656,

heap: 3728384,

nonheap.committed: 62848,

nonheap.init: 24000,

nonheap.used: 62156,

nonheap: 133120,

threads.peak: 18,

threads.daemon: 6,

threads.totalStarted: 176,

threads: 16,

classes: 10294,

classes.loaded: 10294,

classes.unloaded: 0,

gc.ps\_scavenge.count: 11,

gc.ps\_scavenge.time: 405,

gc.ps\_marksweep.count: 0,

gc.ps\_marksweep.time: 0,

datasource.primary.active: 0,

datasource.primary.usage: 0,

counter.status.200.autoconfig: 1,

counter.status.200.beans: 1,

counter.status.200.configprops: 1,

counter.status.200.dump: 1,

counter.status.200.env: 1,

counter.status.200.health: 1,

counter.status.200.info: 1,

counter.status.200.mappings: 1,

gauge.response.autoconfig: 81,

gauge.response.beans: 15,

gauge.response.configprops: 105,

gauge.response.dump: 76,

gauge.response.env: 4,

gauge.response.health: 43,

gauge.response.info: 1,

gauge.response.mappings: 4

}

#### 返回数据示例：/shutdown

要真正生效，得配置文件开启

endpoints.shutdown.enabled: true

#### 返回数据示例：/trace

记录最近100个请求的信息

[{

"timestamp": 1452955704922,

"info": {

"method": "GET",

"path": "/metrics",

"headers": {

"request": {

"Accept - Encoding": "gzip, deflate, sdch",

"Upgrade - Insecure - Requests": "1",

"Accept - Language": "zh-CN,zh;q=0.8,en;q=0.6",

"User - Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36",

"Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8",

"Connection": "keep-alive",

"Host": "localhost:8080"

},

"response": {

"Content - Type": "application/json; charset=UTF-8",

"X - Application - Context": "application:8080",

"Date": "Sat, 16 Jan 2016 14:48:24 GMT",

"status": "200"

}

}

}

}, {

"timestamp": 1452951489549,

"info": {

"method": "GET",

"path": "/autoconfig",

"headers": {

"request": {

"Accept - Encoding": "gzip, deflate, sdch",

"Upgrade - Insecure - Requests": "1",

"Accept - Language": "zh-CN,zh;q=0.8,en;q=0.6",

"User - Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36",

"Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8",

"Connection": "keep-alive",

"Host": "localhost:8080"

},

"response": {

"Content - Type": "application/json; charset=UTF-8",

"X - Application - Context": "application:8080",

"Date": "Sat, 16 Jan 2016 13:38:09 GMT",

"status": "200"

}

}

}

}]

# 总结

Spring Boot是新一代Spring应用的开发框架，它能够快速的进行应用开发，让人忘记传统的繁琐配置，简化了配置、部署、监控的过程，让开发人员更加专注于业务逻辑。

现在Spring官方文档中所有的[Guide](http://spring.io/guides)中的例子都是使用Spring Boot进行构建，这也是一个学习Spring、Spring Boot非常好的地方。

目前，Spring Boot还不完美，还需要与外围的框架集成，例如：服务发现和注册、认证鉴权、监控等方面。

综上，Spring Boot是一个微服务框架的起点，并不是一个完整的解决方案。