第七章 SpringBoot 数据访问

7.1 Mybatis

7.1.1简述Mybatis

Mybatis是当前Java Web开发中流行的持久化ORM框架，它对JDBC进行了封装和简化，无需JDBC的注册驱动，创建Connection连接，配置Statement等繁琐过程，大大减少了JDBC代码，使得开发者只需专注于SQL语句设计即可。Mybatis通过内部机制将Java类（对象）持久化为数据库表中的记录，反之也可以将数据库中的记录转化为Java类（对象）。在三层架构开发中，Mybatis作用在数据访问层，它让数据访问层的开发变得简单，高效。

7.1.2 ORM框架原理

Java程序常常要连接并操作数据库，但是两者的数据类型往往并不匹配，Java是面向对象的语言，Java语言中操作的单元是类与对象，而数据库的数据格式是关系类型。为了匹配Java面向对象与关系型数据库的数据类型，人们发明了ORM框架（Object Relational Mapping，对象关系映射）,用于将Java中的对象映射成数据库中的记录，对象中的属性映射为数据库中的字段，程序员可以使用面向对象的编程方式来操作数据库。例如，在程序代码中添加一个对象，则数据库中相应添加一行记录，在程序代码中删除一个对象，则数据量中相应删除了一条记录。比较常见的ORM框架有Hibernate和Mybatis。

ORM框架的原理如图7.1所示。其中，CRUD是在做计算处理时的增加（Create），读取查询（Retrieve），更新（Update）和删除（Delete）4个单词的首字母缩写，POJO(Plain Ordinary Java Object)为简单的Java对象，实际就是普通的JavaBean。

数据库表

ORM框架

POJO对象

Java程序

图7.1

7.1.3 SpringBoot使用Mybatis

此处通过演示一个实例，在MySQL数据中的user表中增加和查询用户信息。同时为了简化实例，省略了service层，将具体操作整合在了Controller中。

1.导入Mybatis相关依赖

<dependency>

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

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

<version>x.x.x</version>

</dependency>

MyBatis-Spring-Boot-Starter依赖将会提供：自动检测现有的DataSoure，创建并注册SqlSessionFactory的实例，该实例使用SqlSessionFactoryBean将该DataSource作为输入进行传递，创建并注册从SqlSessionFactory中获取的SqlSessionTemplate的实例。自动扫描项目的mappers，将它们连接到SqlSessionTemplate并将其注册到Spring上下文，以便将它们注入到项目的bean中。使用了该Starter之后，只需要定义一个DataSource即可（application.properties或application.yml中可配置），它会自动创建使用该DataSource的SqlSessionFactoryBean以及SqlSessionTemplate，也会自动扫描项目的Mappers，连接到SqlSessionTemplate，并注册到Spring上下文中。

2.在application.yml中配置数据库连接信息

数据库配置

spring:

datasource:

username: root

password: 123456

url:jdbc:mysql://localhost:3306/springboot?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8

driver-class-name: com.mysql.cj.jdbc.Driver

整合mybatis

mybatis:

type-aliases-package: com.example.pojo

mapper-locations: classpath:mybatis/mappper/\*.xml

mapper-locations: 是指mapper的位置，也就是.xml文件的位置，如果没有配置，就会默认去mapper接口类所在的包中查找。但是注意，如果在编译之后，Java文件夹和resources文件夹的文件都会到同一个包中。

type-aliases-package:实体类的别名。这个包下面的实体类都可以使用别名，例如：

com.example.pojo.User 可以用User代替

3.在com.example.pojo包下创建一个实体类User

package com.example.pojo;

public class User {

private int id;

private String name;

private String gender;

//省略了getter，setter方法

}

4.创建mapper包以及和实体类对应的Usermapper接口

package com.example.mapper;

@Mapper//这个注解表示这是一个mybatis 的mapper类，不可省略。

@Repository

public interface UserMapper {

List<User> queryUserList();

User queryUserById(int id);

int addUser(User user);

int deleteUser(int id);

int updateUser(User user);

User selectUserById(int id);

}

5.编写mapper接口的映射文件

Mapper接口的映射文件可以和mapper接口放在同一个包中，但是为了方便管理，可以将映射文件放到resources资源文件夹中，在resources目录下新建一个mapper文件，然后新建一个UserMapper.xml文件。

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

命名空间

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.UserMapper">

SQL语句查询结果映射为User类型。注意，如果在整合Mybatis时没有设置别名，则需要写全称类名

<select id="queryUserList" resultType="User">

select \* from user

</select>

<select id="selectUser" resultType="User">

select \* from user

</select>

<select id="selectUserById" resultType="User">

select \* from user where id = #{id}

</select>

<insert id="addUser" parameterType="User">

insert into user (id,name,gender) values (#{id},#{name},#{gender})

</insert>

<update id="updateUser" parameterType="User">

update user set name=#{name},pwd=#{gender} where id = #{id}

</update>

<delete id="deleteUser" parameterType="int">

delete from user where id = #{id}

</delete>

</mapper>

映射文件主要实现SQL语句与Java对象之间的映射，使得SQL语句查询出来的关系型数据能够封装成Java对象。映射文件通常用POJO+Mapper命名。

映射文件中的一级标签<mapper></mapper>里面可以包含多对<select>标签或者<insert>等其他标签。<mapper>标签的namespace属性，用于标识映射文件，通常其值设置成对应接口的全路径名称。

6.在controller包中编写UserController类，

package com.example.controller;

@RestController

public class UserController {

@Autowired

private UserMapper userMapper;

//查询所有用户

@GetMapping("/queryUserList")

public List<User> queryUserList(){

List<User> usersList = userMapper.queryUserList();

for (User user:usersList){

System.out.println(user);

}

return usersList;

}

//根据id选择用户，并在控制台输出

@GetMapping("/selectUserById")

public void selectUserById(){

User user = userMapper.selectUserById(5);

System.out.println(user);

}

//添加一个用户

@GetMapping("/addUser")

public String addUser(){

userMapper.addUser(new User(5,"小白","女"));

return "ok";

}

}

在浏览器中现后输入http://localhost:8080/addUser和http://localhost:8080/selectUserById。可以在控制台看到如图7.2的内容



图7.2

7.1.4 Mybatis注解方式开发

Springboot整合Mybatis注解方式开发，可以减少让我们的开发变得更加简单，不需要去配置Mapper.XML，。本节将用一个实例来解释如何在springboot使用mybatis注解开发，为了简便，省略了service层。

注意：注解方式开发发无需添加新的依赖。

1. 在application.yml中配置数据源信息

数据库连接信息和上一节完全相同，此处不再赘述。

整合mybatis

mybatis:

type-aliases-package: com.example.pojo

~~mapper-locations: classpath:mybatis/mappper/\*.xml~~

此处不再需要配置mapper-locations，因为在注解方式开发下不再需要对应mapper.xml

1. 新建一个pojo包并新建一个User类

public class User {

private int id;

private String name;

private String gender;

public User() {

}

public User(int id, String name, String gender) {

this.id = id;

this.name = name;

this.gender = gender;

}

//省略getter（）和setter方法

}

1. 创建mapper包以及和实体类对应的Usermapper接口

@Mapper

@Repository

public interface UserMapper {

//查询所有用户

@Select("select id,name,gender from user")

List<User> getAllUser();

//根据id查询用户

@Select("select \* from user where id = #{id}")

User selectUserById(Integer id);

//添加一个用户

@Insert("insert into user(id,name, gender) values(#{id},#{name}.#{gender})")

int addUser(User user);

}

mapper类中在对应的方法中添加@select()，@Insert()，@update()，@delete()的注解，将SQL语句写入到对应的注解中，就不需要去\*\*\*mapper.xml映射文件中配置SQL等信息了。

1. 在controller包中编写UserController类

@RestController

public class UserController {

@Autowired

private UserMapper userMapper;

@GetMapping("getAll")

public List<User> getAllUser(){

List<User> userList = userMapper.getAllUser();

return userList;

}

@GetMapping("/addUser")

public void addUser(){

userMapper.addUser(new User(6,"小红","女"));

}

@GetMapping("selectUserById")

public User selectUserById(){

User user = userMapper.selectUserById(6);

return user;

}

}

Controller层和上一节编写方式没有区别。

1. 测试

运行程序，然后在浏览器先后输入localhost:8080/addUser和localhost:8080/ selectUserById 可以看到如图7.3所示内容



图7.3

至此测试成功。

7.2 MongoDB

7.2.1 简述

MongoDB 是由C++语言编写的，是一个**基于分布式文件存储**的开源数据库系统。它旨在为WEB应用提供可扩展的高性能数据存储解决方案，MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组。它作为一种NoSQL存储介质，存储读取快，但实质上它又是一种介于关系型数据库（如mysql，数据存在磁盘中）和非关系型数据库（如redis，数据存在内存中）的介质，它数据存在磁盘，但读取又借助内存机制映射进行，所以集成了关系型和非关系型的各自优点。

7.2.2配置MongoDB

从官方网站[https://www.mongodb.com](https://www.mongodb.com/download-center" \l "community) 中下载MongoDB。此处演示下载的是MongoDB4.2版本。安装后目录如下：

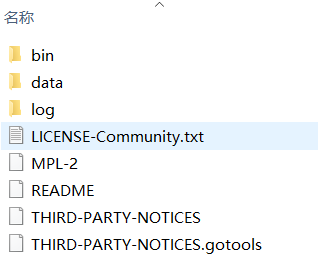


图7.2.1 MongoDB文件结构

配置并从配用路径开启：

在安装目录\data\下创建一个db目录，作为安装目录\data\的备用目录，在log文件夹下创建新文件 mongo.config(安装目录\log\mongo.config)

在启动时，使用的是安装目录\bin\下的mongod.exe，如果直接启动，实际使用的是C:\data\db\为数据库文件存储目录，需要确保该目录存在。

可以使用--dbpath "安装目录\data\db"显式指定该目录为存储位置，来启动MongoDB，如：

D:\ProgramFiles\MongoDB\Server\4.2\bin>mongod.exe

--dbpath "D:\Program Files\MongoDB\Server\4.2\data\db"

测试是否成功启动服务，MongoDB默认的监听端口是：27017。在浏览器中输入<http://localhost:27017/，若页面出现> It looks like you are trying to access MongoDB over HTTP on the native driver port. 则表明服务已经启动

测试使用：使用安装目录\bin\下的mongo.exe 启动MongoDB的操作终端：

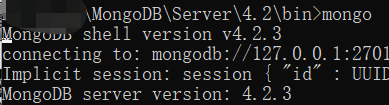


图7.2.2操作演示

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>

测试使用成功。

7.2.3基本用法

创建数据库：use database\_name

实例：

> use first

switched to db firstdb

查看当前正在使用的数据库：db

> db

firstdb

>

查看所有数据库：show dbs

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>

此时新建的firstdb不在数据库的列表中，只有当集合内容插入到数据库firstdb中才会正在被创建。

注意：刚开始创建的数据库是存在内存中的，真正创建后数据库被存在磁盘中。 MongoDB 中默认的数据库为 test，如果你没有创建新的数据库，集合将存放在 test 数据库中。

删除数据库：db.dropDatabase()

删除当前数据库，默认为 test，可以先使用 db 命令查看当前数据库名，再决定是否执行删除命令。

集合的创建

当向一个集合中插入内容的时候，如果集合不存在则为自动创建。

此外，也可以使用显式创建集合的方式

基本语法格式：db.createCollection(name，options)

参数说明：

name ：要创建的集合名称

options: 可选参数, 指定有关内存大小及索引的选项

删除集合：db.collection.drop()

collection就是我们想删除集合名字，执行完命令后会出现返回值：

删除成功，返回值是true；否则返回false

例如：删除firstacollection集合

执行：db.firstcollection.drop()

查看当前数据库中的表：show collections或show tables

插入文档

注意：存储再集合中的数据都是bson格式，bson数据结构和json基本一致。

语法：db.collection\_name.insert(document) 或者 db.col.save(document)

document为所所要插入的文档内容，格式是bson。

更新文档

MongoDB 使用 update() 和 save() 方法来更新集合中的文档，

update()方法

语法格式：db.collection\_name.update(

<query>,

<update>,

{

upsert: <boolean>,

multi: <boolean>,

writeConcern: <document>

}

)

参数说明：

query : update的查询条件，类似sql update查询内where后面的。

update : update的对象和一些更新的操作符（如$,$inc...）等，可以理解为sql update查询内set后面的

upsert : 可选，它表示如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。

multi : 可选，mongodb 默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

writeConcern :可选，抛出异常的级别。

save方法

此方法通过传入文档来替换已有文档。

语法格式：db.collection\_name.save(

<document>,

{

writeConcern: <document>

}

)

参数说明：

document : 文档数据。

writeConcern :可选，抛出异常的级别

删除文档

删除文档使用remove函数

db.collection\_name.remove(

<query>,

{

justOne: <boolean>,

writeConcern: <document>

}

)

参数说明：

query :（可选）删除的文档的条件。

justOne : （可选）如果设为 true 或 1，则只删除一个文档，若不设置该参数，或使用默认值 false，则删除所有匹配条件的文档。

writeConcern :（可选）抛出异常的级别。

若要删除所有数据，则执行以下操作

db.collection\_name.remove({ })

查询文档

查询文档使用find()方法，此方法以非结构化的方式来显示所有文档。

语法格式：db.collection\_name.find(query, projection)

参数说明：

query ：可选，使用查询操作符指定查询条件

projection ：可选，使用投影操作符指定返回的键。查询时返回文档中所有键值， 只需省略该参数即可（默认省略）。

7.2.4 SpringBoot使用MongoDB

Spring Boot中可以通过在pom.xml中加入spring-boot-starter-data-mongodb引入对mongodb的访问支持依赖

<dependency>

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

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

</dependency>

配置数据源，在application.yml中配置

本地安装的mongodb，没有用户名和密码

spring:

data:

mongodb:

uri: mongodb://localhost:27017/test

有用户名和密码的写法

spring:

data:

mongodb:

uri: mongodb://name:password@localhost:27017/test

MongoDB存储的是bson数据，所以需要创建一个实体类与MongoDB存储的bson数据形成对应，即bson转成实体对象，实体对象可以转换成bson的中间数据模型。

此处创建一个MongodbTest作为实体类

public class MongodbTest {

private Integer id;

private Integer age;

private String name;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public String getName() {

return name;

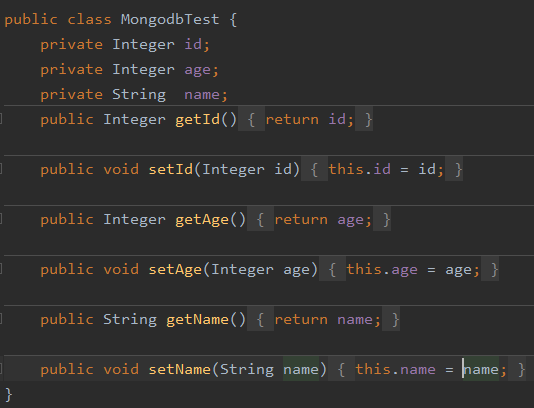
}

public void setName(String name) {

this.name = name;

}

}



在dao层中创建一个MongodbTestDao类

@Component

public class MongodbTestDao {

@Autowired

private MongoTemplate mongoTemplate;

/\*

创建对象

\*/

public void saveTest(MongodbTest test){

mongoTemplate.save(test);

}

/\*\*

\* 根据用户名查询对象

\* @param name

\* @return

\*/

public MongodbTest findTestByName(String name){

Query query = new Query(Criteria.where("name").is(name));

MongodbTest mgt = mongoTemplate.findOne(query,MongodbTest.class);

return mgt;

}

}

此处作为一个简单实例，省略了service层，相关操作在Controller中进行。

在Controller层中新建MongodbForTest类

@RestController

public class MongodbForTest {

@Autowired

private MongodbTestDao mtdao;

@RequestMapping(value = "/test1")

public void saveTest() throws Exception{

MongodbTest mgtest = new MongodbTest();

mgtest.setId(2);

mgtest.setAge(33);

mgtest.setName("Jony");

mtdao.saveTest(mgtest);

}

@RequestMapping(value = "/test2")

public MongodbTest findTestByName(){

MongodbTest mgtest = mtdao.findTestByName("Jony");

System.out.println("mgtest is"+mgtest);

return mgtest;

}

}

运行程序，再浏览器输入http://localhost:8080/test1

结束后,重新输入<http://localhost:8080/test2>，可以看到浏览器中出现下面的数据：

{"id":2,"age":33,"name":"Jony"}

则表明程序执行成功。

7.3Redis

7.3.1 Redis简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。它支持多种类型的数据结构，如字符串（strings），散列（hashs），列表（lists），集合（sets），有序集合（sorted sets）。这些数据类型都支持push，pop，add，remove以及取交集并集和差集等丰富的操作。在此基础上，Redis支持各种不同方式的排序。

Redis支持数据的持久化同时也具有很高的效率，它的数据都是缓存在内存中，同时会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件。它的操作都是原子性的，所谓原子性就是指要么成功执行操作，要么失败完全不执行操作。它也有丰富的特性，可以用于缓存，实现发布/订阅机制，按key设置过期时间，过期后将会自动删除。

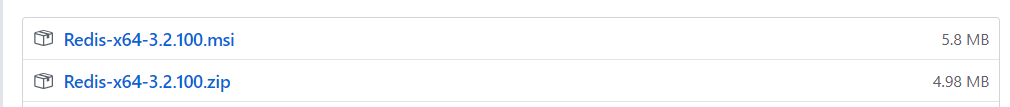
不过需要注意的是，Redis运行在内存中，所以在对不同数据集进行读写时需要权衡内存的大小，因为数据量不能大于内存。

7.3.2 Redis安装与配置

1.window10上安装Redis

可从Redis官网：<https://redis.io/> 上下载对应的版本，

或者从github：https://github.com/MicrosoftArchive/redis/releases下载。Redis支持32位和64位，根据自身系统平台进行选择。



msi后缀的文件是安装版，zip后缀的文件是解压版。

此处下载msi版本，下载后打开文件，一路点击next，直到出现图7.3.1

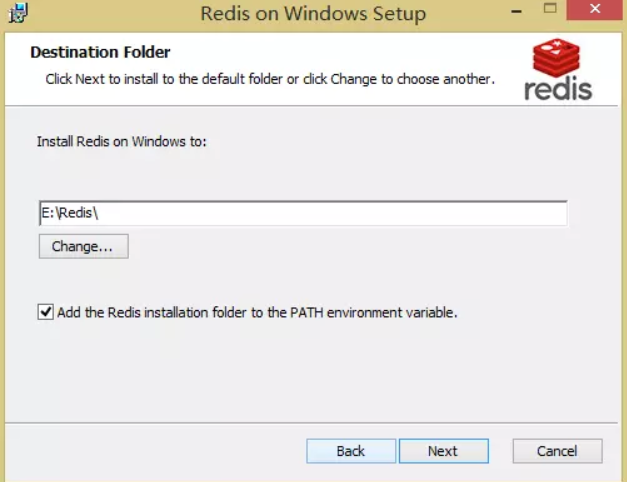


图7.3.1

选择文件安装位置，勾选上添加安装目录到 PATH 环境变量的选项。继续点击next，出现图7.3.2

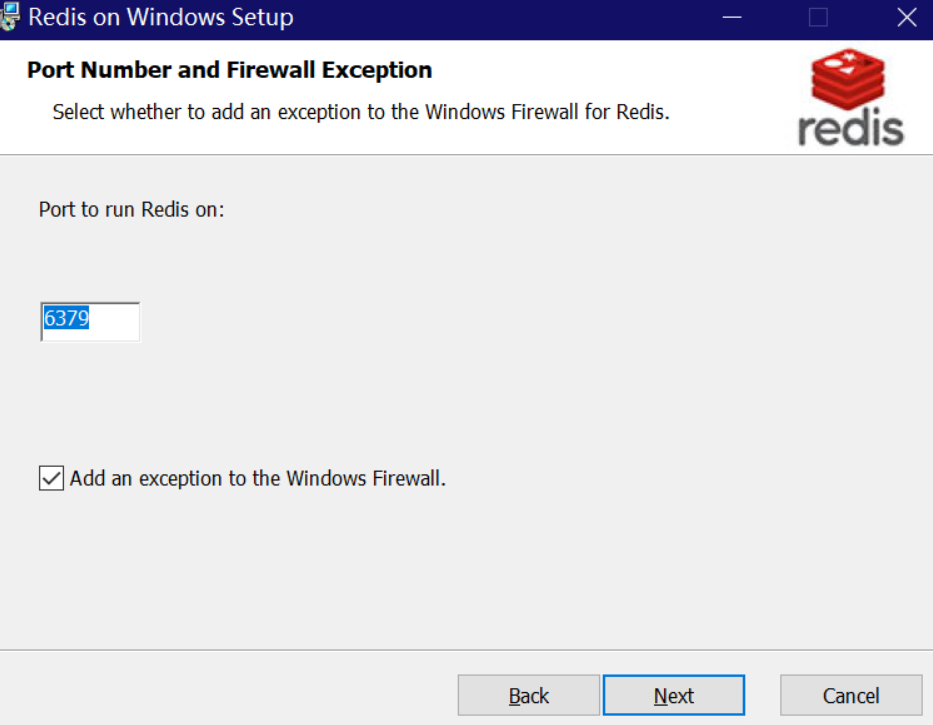


图7.3.2

选择端口，Redis默认是6739端口，若不想用6379或者6379端口已经被占用，可以修改成其他端口。下方是添加到windows防火墙的选项，默认勾选。继续点击next。出现图7.3.3

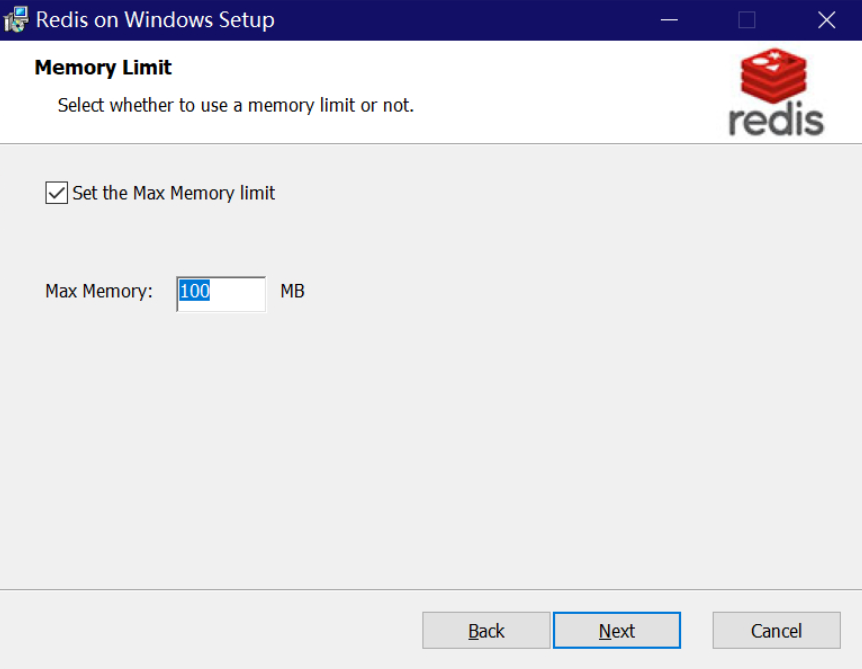


图7.3.3

此页面是设置Redis最大内存限制，默认是100MB内存，根据需要可以修改大小。继续点击next，然后点击Install，等到片刻，点击finsh，安装完成。

2.测试安装的Redis

若是采用msi文件的安装

第一种方式，可以从计算机管理->服务与应用->服务 查看Redis是否正在运行。如图7.3.4

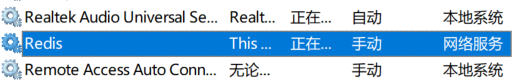


图7.3.4

若正在运行，则说明Redis服务启动成功。

第二种方式，可以在命令行窗口进入到安装路径的根目录。

输入命令redis-server redis.windows.conf，若出现图7.3.5所示内容，则说明服务启动成功。

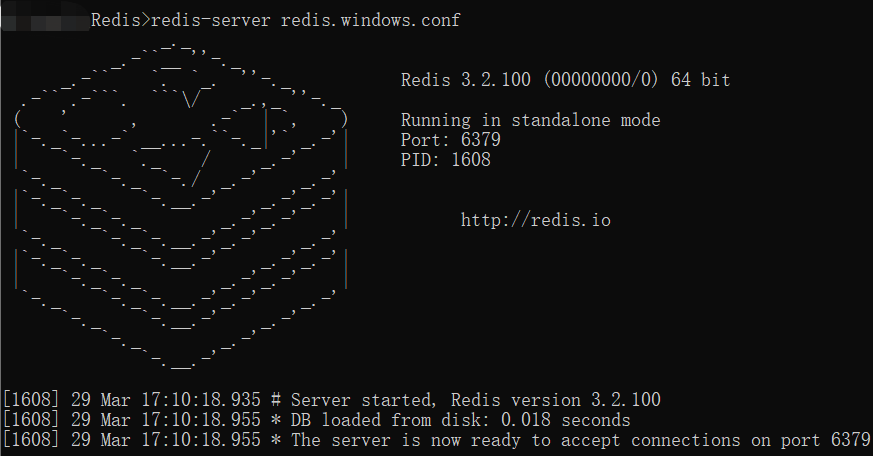


图7.3.5

测试使用（注意必须在启动服务后才能使用），可以从Redis的安装根目录中找到redis-cli.exe文件，点击启动。设置键值对，进行测试。例如图7.3.6

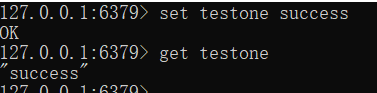


图7.3.6

至此，测试成功。

7.3.3 Redis数据类型

1.字符串（string）类型

字符串类型是 Redis 中最基本的数据类型，它能存储任何形式的字符串，包括二进制数据，序列化后的数据，JSON 化的对象甚至是一张图片。

|  |  |
| --- | --- |
| key | value |
| Age | 18 |

2.哈希（hash）类型

hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。Redis 中每个 hash 可以存储 232 - 1 键值对。

|  |  |
| --- | --- |
| key | user |
| field | value |
| name | 小明 |
| age | 18 |

3.列表（list）类型

Redis 列表是简单的字符串列表，按照插入顺序排序，可以通过添加一个元素到列表的头部（左边）或者尾部（右边）。

|  |  |
| --- | --- |
| key | value |
| students | 小明 小红 小王 |

4.集合（set）类型

Redis 的 集合类型 是 string 类型的无序集合，集合成员是唯一的，即集合中不能出现重复的数据。

|  |  |
| --- | --- |
| Key | Value |
| Students | 小明 |
| 小红 |
| 小王 |

6.有序集合（sorted set）类型

Redis 有序集合和集合一样也是 string 类型元素的集合，且不允许重复的成员。不同的是 有序集合的每个元素都会关联一个分数（分数可以重复），redis 通过分数来为集合中的成员进行从小到大的排序。

|  |  |
| --- | --- |
| Key | Value |
| Students | 小明 5000 |
| 小红 6000 |
| 小王 7000 |

7.3.4 Redis命令

1.基本操作命令

redis 默认为 16 个库 ，redis 默认自动使用 0 号库。

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| **ping** | Pong | 检查Redis服务状态，返回pong表示服务运行正常 |
| **dbsize** | 数字 | 查看当前数据库中 key 的数目,返回值表示当前数据库中key的数量 |
| **select** db | OK | 切换数据库命令 |
| **flush** db | OK | 删除当前库的数据 |
| **exit**或**quit** |  | 退出当前Redis连接 |

2.key的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| keys **pattern** |  | 查找所有符合模式pattern的key。 |
| **exists** key[key…] | 存在key返回1，否则返回0,使用多个key，返回存在key的数量。 | 判断key是否存在。 |
| **expire** key seconds | 设置成功返回数字 1，否则返回1。 | 设置 key 的生存时间，超过时间，key 自动删除，单位是秒。 |
| **ttl** key | Key永不过期返回-1，key不存在返回-2，其他数字表示key的剩余时间 | 查看key的剩余生存时间 |
| **type** key | 字符串表示的数据类型 | 查看key所存储值的数据类型 |
| **del** key [key…] | 数字，表示删除的key的数量 | 删除存在的key，不存在的key忽略 |

3.字符串类型（string）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| set key value |  | 将字符串值value设置到key中 |
| get key | 返回key对应的value | 获取key中设置的字符串值 |
| incr key | 数字 | 将key中存储的数据值加1，若key不存在，则key先被初始化为0，再加1，注意：只能对数字类型的数据操作 |
| decr key | 数字 | 将key中存储的数据值减1，若key不存在，则key先被初始化为0，再减1，注意：只能对数字类型的数据操作 |
| append key value | 追加字符串之后的总长度 | 如果 key 存在，则将 value 追加到 key 原来值的末尾；如果 key 不存在，则将 key 设置值为 value |
| strlen key | 若key存在，返回字符串值的长度，否则返回0 | 查询key所储存的字符串值的长度 |
| getrange key start end | 截取的自字符串 | 获取 key 中字符串值从 start 开始 到 end 结束 的子字符串,包括 start 和 end, 负数表示从字符串的末尾开始， -1 表示最后一个字符 |
| setrange key offset value | 修改后的字符串的长度 | 用 value 覆盖key 的存储的值从 offset 开始,不存在的 key 做空白字符串。 |
| mset key value [key value…] | OK | 同时设置一个或多个 key-value 对返回值 |
| mget key [key …] | 包含所有 key 的列表 | 获取所有(一个或多个)给定 key 的value |

4.哈希类型（hash）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| hset hash key field value | 若field是hash表中的新field，设置成功返回1；若field已经存在，旧值覆盖新值，返回0 | 将哈希表 key 的域 field 的值设为 value ，如果 key 不存在，则新建 hash 表，执行赋值，如果有 field ,则覆盖值。 |
| hget key field | 若存在，则返回field域的值；若key或者field不存在，则返回nil | 获取哈希表中key给定域field的值 |
| hmset key field value [field value…] | 设置成功返回OK；失败返回错误 | 将多个 field-value (域-值)设置到哈希表 key 中，会覆盖已经存在的 field，若hash 表 key 不存在，创建空的 hash 表，执行 hmset |
| hmget key field [field…] | 返回与field顺序对应的值；若相应位置的field不存在,返回nil | 查询哈希表 key 中一个或多个给定域的值 |
| hgetall key | 以列表形式返回 hash 中域和域的value ；若key 不存在，返回empty list or set | 查询哈希表key中的所有域和值 |
| hdel key field [field…] | 成功删除的field的数量 | 删除哈希表 key 中的一个或多个指定域 field |
| hkeys key | 包含所有 field 的列表，key 不存在返回空列表 | 查看哈希表key中所有field |
| hvals key | 哈希表中所有域的值；key不存在则返回空列表 | 查询哈希表中所有域的值 |
| hexists key field | 若存在返回值是1；否则返回0 | 查看哈希表key中，给定域field是否存在 |

6.列表（list）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| lpush key value [value…] | 新列表的长度 | 将一个或多个 value 插入到列表 key 的表头（最左边），从左到右的顺序依次插入到表头 |
| rpush key value [value…] | 新列表的长度 | 将一个或多个 value 插入到列表 key 的表尾（最右边），从左到右的顺序依次插入到表尾 |
| lrange key start stop | 指定区间的列表 | 获取列表 key 中指定区间内的元素 |
| lindex key index | 指定下标的元素；index不在列表范围内，返回nil | 查询列表 key 中下标为指定 index 的元素 |
| llen key | 列表的长度；key不存在返回0 | 查询列表key的长度 |
| lrem key count value | 移除的元素的个数 | 移除列表中与参数 value 相等的元素， count >0 ，从列表的左侧向右开始移除； count < 0 从列表的尾部开始移除；count = 0 移除表中所有与 value 相等的值。 |
| lset key index value | 设置成功返回OK，否则返回错误信息 | 设置key下表为index的元素的值为value |
| linsert key BEFORE|ALFTER pivot value | 若命令成功执行，返回新列表的长度；若不存在pivot，返回-1；若key不存在，返回0 | 将value插入到列表key中位于值 pivot 之前或之后的位置；若key不存或者pivot不在列表中，不执行任何操作。 |

7.集合类型（set）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| sadd key member [member…] | 加入到集合的新元素的个数 | 将一个或多个元素加入到集合key中，已经存在的元素被忽略 |
| smembers key | 集合中成员的列表 | 查询集合中所有成员元素，不存在的key被视为空集合 |
| sismember key member | 是集合的成员返回1；否则返回0 | 检查元素是否是集合里面的成员 |
| scard key | key的元素个数；不存在的key则返回0 | 查询集合里面的元素个数 |
| srem key member [member…] | 成功删除元素的个数 | 删除集合key中的一个或多个成员，不存在的元素被忽略 |
| srandmember key [count] | 一个元素或者多个元素的集合 | 返回集合中一个或多个随机元素 |
| spop key [count] | 被删除的元素；key不存在或者空集合则返回nil | 从集合中随机删除一个或者删除指定个数的元素, count 是删除的元素个数 |

8.有序集合类型（sorted set）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| zadd key score member [score member…] | 新添加元素的个数 | 将一个或多个元素及其score值加入到有序集合 key中 |
| zrange key start stop [WITHSCORES] | 自定区间的成员集合 | 查询有序集合内指定区间的元素 |
| zrevrange key start stop [WITHSCORES] | 自定区间的成员集合 | 查询有序集合key内指定区间的元素 |
| zrem key member [member…] | 删除成功的元素数量 | 删除有序集合key中的一个或多个成员 |
| zcard key | Key存在则返回集合元素的个数；key不存在则返回0 | 查询有序集合key的元素个数 |
| zcount key min max | 返回成员数量 | 查询有序集合key中，score值在min和max之间的成员数量； |

7.3.5 SpringBoot使用Redis

要在Springboot项目中使用Redis，需要在pom.xml中添加spring-boot-starter-data-redis依赖，并在application.properties中配置相关属性，配置结束后就可以使用StringRedisTemplate模板类来操作Redis。StringRedisTemplate继承了RedisTemplate，它封装了对Redis的一些常用操作，但是只能对key和value都是String的键值对进行操作。

下面将通过一个实例来介绍在Spring Boot中如何使用Redis来实现储存字符串和对象。该项目是继续对上一个小节中项目的扩展。

1.在pom.xml中添加spring-boot-starter-data-redis依赖。

<!--Redis依赖-->

<dependency>

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

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

</dependency>

2.在application.properlies中添加Redis相关配置

# Redis数据库索引（默认为0）

spring.redis.database=0

# Redis服务器地址

spring.redis.host=127.0.0.1

# Redis服务器连接端口

spring.redis.port=6379

# Redis服务器连接密码（默认为空）

spring.redis.password=

# 连接池最大连接数（使用负值表示没有限制）

spring.redis.jedis.pool.max-active=-1

# 连接池最大阻塞等待时间（使用负值表示没有限制）

spring.redis.jedis.pool.max-wait=-1

# 连接池中的最大空闲连接

spring.redis.jedis.pool.max-idle=8

# 连接池中的最小空闲连接

spring.redis.jedis.pool.min-idle=0

3.新建Utils工具包并新建JsonUtils类

public class JsonUtils {

//定义Json对象

private static final ObjectMapper MAPPER=new ObjectMapper();

/\*\*

\* 将对象转换成Json字符串

\* @param data

\* @return

\*/

public static String objectToJson(Object data){

try {

String string = MAPPER.writeValueAsString(data);

return string;

}catch (JsonProcessingException e){

e.printStackTrace();

}

return null;

}

public static <T> T jsonToPojo(String jsonData,Class<T>beanType){

try {

T t = MAPPER.readValue(jsonData,beanType);

return t;

}catch (Exception e){

e.printStackTrace();

}

return null;

}

}

4.在controller层的UserController类中添加下面两个方法

@RequestMapping("/setUserPojo")

public void setUserPojo(){

User user = new User(10, "小明", "男");

stringRedisTemplate.opsForValue().set("jsonUser", JsonUtils.objectToJson(user));

}

@RequestMapping("/getUserPojo")

public User getUserPojo( ){ User user =

JsonUtils.jsonToPojo(stringRedisTemplate.opsForValue().get("jsonUser"),User.class);

return user;

}

5.运行程序

在浏览器中先输入<http://localhost:8080/setUserPojo>；

后输入<http://localhost:8080/getUserPojo>，会看到页面中有如下的内容：

{"id":10,"name":"小明","gender":"男"}

说明项目运行成功。

6.总结

Redis的使用场景十分丰富，以上的例子仅仅只是冰山一角，Redis可以用作全页缓存，会话缓存（Session Cache），快速计数，限速，消息队列，发布订阅等等。更多关于Redis的知识不再此作详细介绍。

# SpringBoot Actuator

## SpringBoot Actuator 简介

Spring Boot 包含许多其他功能，可帮助你在将应用程序推送到生产环境时监控和管理应用程序，你可以选择使用 HTTP 端点或 JMX 来管理和监控应用程序，审计、健康和指标收集也可以自动应用于你的应用程序。Spring Boot Actuator便可以帮助我们全方位的监控应用节点，它提供了众多HTTP接口端点（Endpoint），里面包含了丰富的应用程序运行时的内部状态信息，我们可以有选择的开启和关闭部分功能。我们只需要导入Actuator模块对应的依赖组件spring-boot-starter-actuator，就可以使用Actuator了。

## 原生端点使用

1.导入依赖组件

首先，需要在项目中导入spring-boot-starter-actuator的依赖

<dependency>

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

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

</dependency>

启动应用，访问<http://localhost:8080/actuator>，观察actuator提供了哪些接口。

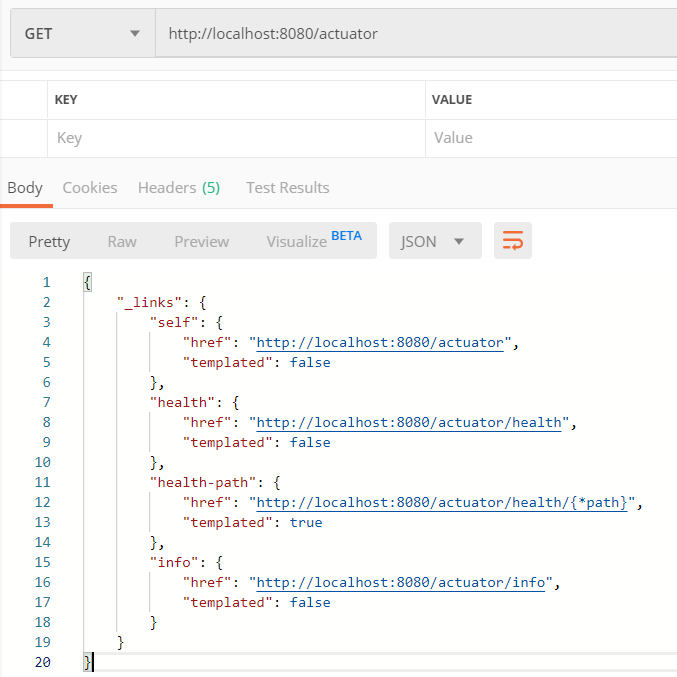


图8-1

上图中可以看到显示了三个接口，但是这只是默认显示，可以通过配置管理显示更多接口。

我们可以了解一些常见的endpoint，下面是部分常见endpoint介绍，更多的endpoint可以到spring官网中查询。

|  |  |
| --- | --- |
| Endpoint ID | Description |
| info | 显示应用的基本信息 |
| health | 显示应用的健康状态 |
| metrics | 显示应用的度量信息 |
| loggers | 显示和修改配置的loggers |
| logfile | 返回日志文件的内容（如果已设置logging.file.name或logging.file.path属性） |
| httptrace | 显示HTTP跟踪信息 |
| mappings | 显示一个所有@RequestMapping路径的集合列表 |
| env | 显示Spring的ConfigurableEnvironment中公开的属性 |

需要注意的是，在springboot配置文件中配置访问根路径，springboot2.0+默认： /actuator，旧版本为/ 。

可以通过在application.properties中配置访问根路径

management.endpoints.web.base-path=/actuator

从图8-1中我们可以看到两个端点health和info。尽管只有两个端点是可以看到的，但是默认情况下，除了shutdown端点是关闭的，其它的都是启用的。配置一个端点的启用，可以通过使用management.endpoint.< Endpoint ID >.enabled。例如：

management.endpoint.shutdown.enabled=true

也可以通过下面的配置来启用所有的端点

management.endpoints.enabled-by-default=true

端点启用不代表端点可以暴露，想要暴露端点需要另外配置如下：

|  |  |
| --- | --- |
| 属性 | 默认 |
| management.endpoints.jmx.exposure.exclude |  |
| management.endpoints.jmx.exposure.include | \* |
| management.endpoints.web.exposure.exclude |  |
| management.endpoints.web.exposure.include | info, health |

**注意：\***表示暴露所有端点，若是在yaml文件中配置，使用\*号必须加引号。

一些端点的介绍如下：

（1）health端点

{

"status": "UP"

}

status值为UP说明应用是健康的，如果应用不健康，将会显示DOWN。若想要看出详细的监控信息，可以在application.properties中增加如下配置：

management.endpoint.health.show-details=always

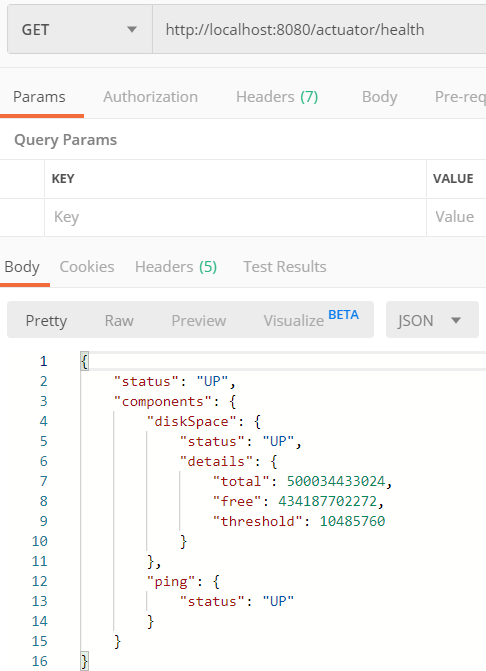


图8-2

（2）metrics



图8-3

metrics端点展示了我们可以追踪的度量，例如：jvm内存，jvm线程，cpu的使用等等，可以通过度量名称传入接口访问对应度量的详细信息。如图8-3

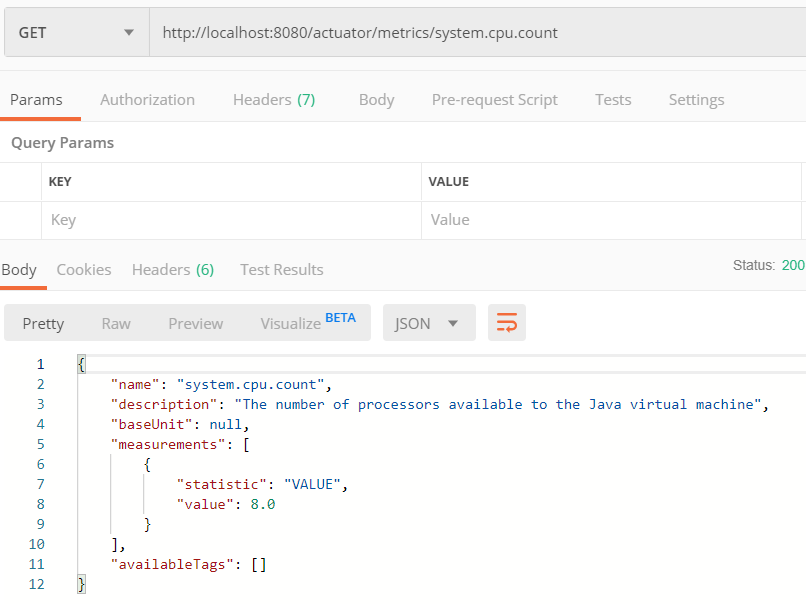


图8-4

（3）loggers

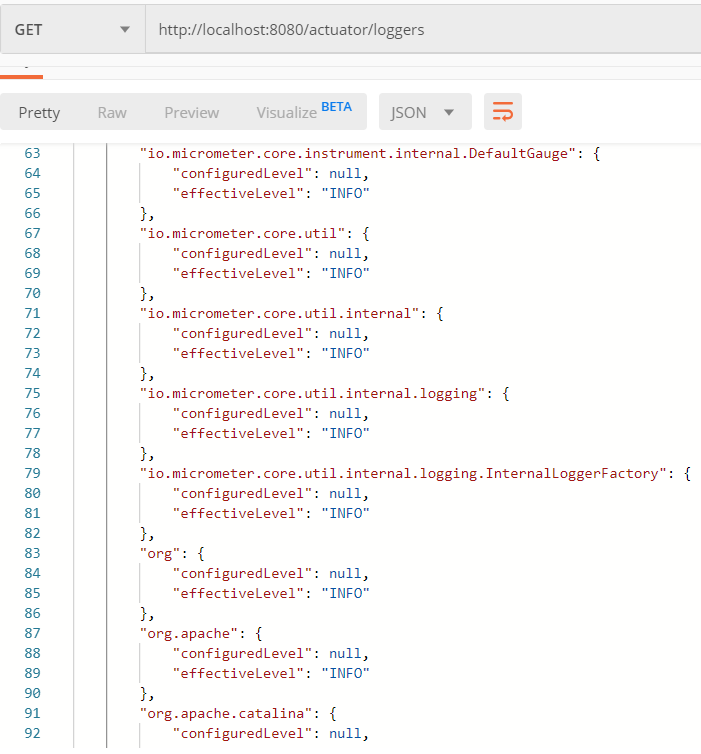


图8-5

loggers展示了应用中可配置的loggers列表和相关的日志等级。可以通过如下配置查看单个logger的日志配置信息。

http://localhost:8080/actuator/loggers/{name}

（4）mappings

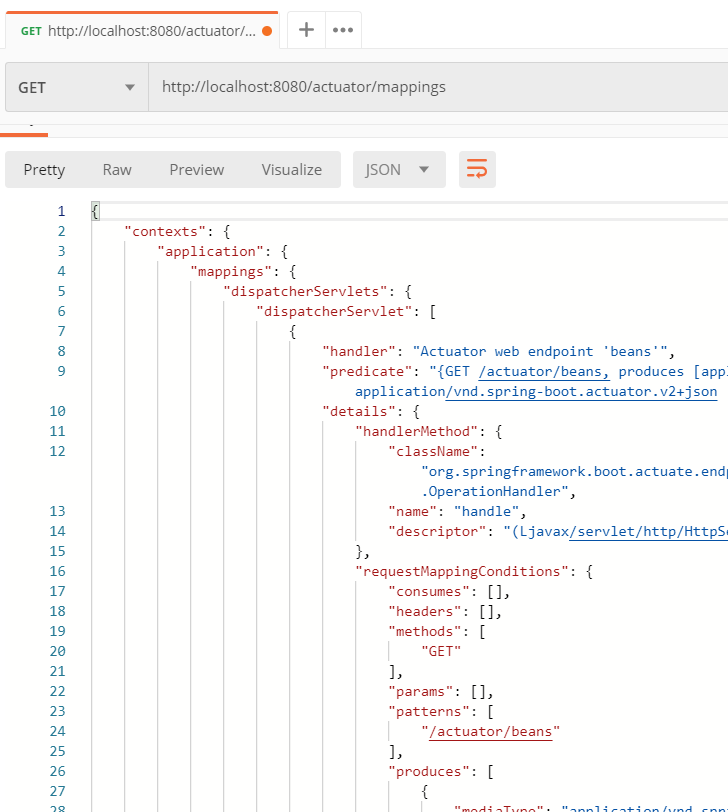


图8-6

mappings端点用于查看所有URL映射，即所有@RequestMapping路径的整理列表。

（5）env

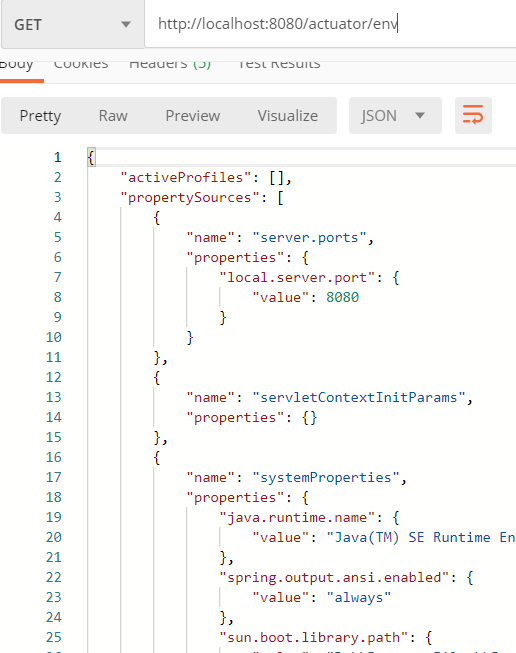


图8-7

env端点可以获取绝大多数关于当前应用程序的运行环境信息，例如JDK版本及ClassPath信息，当前启用的配置文件（activeProfiles），propertySources等。可以让我们了解应用程序的运行状况。

## 定制Actuator

Spring Actuator中原生端点一般可以分为三类。

应用配置类：获取应用程序中加载的应用配置、环境变量、自动化配置报告等与Spring Boot应用密切相关的配置类信息。

度量指标类：获取应用程序运行过程中用于监控的度量指标，比如：内存信息、线程池信息、HTTP请求统计等。

操作控制类：提供了对应用的关闭等操作类功能。

当这些端点无法满足我们的需要时，我们可以自定义端点。

Spring boot 2之前的版本的应用程序通过Endpoint可以根据应用程序业务需求实现自定义的端点，但是其自定义端点需要实现接口中的多个方法，操作起来比较繁琐。Spring boot 2

之后的版本中实现自定义端点代码精简。下面介绍2.0版本后的自定义端点的方法。

在项目中新建一个actuator包，新建一个类CustomEndPoint，代码如下：

@Endpoint(id = "myendpoint")

@Component

public class CustomEndPoint{

@ReadOperation

public Map<String,String> TestOne(){

Map<String,String> map = new HashMap<>();

map.put("name","Mike");

map.put("age","18");

return map;

}

@ReadOperation

public Map<String,String> TestTwo(@Selector String age){

Map<String,String> map = new HashMap<>();

map.put("name","Mike");

map.put("age",age);

return map;

}

@WriteOperation

public Map<String,String> TestThree(@Selector String age){

Map<String,String> map = new HashMap<>();

map.put("name","Jony");

map.put("age",age);

return map;

}

}

注意：①需要配置management.endpoints.web.exposure.include=myendpoint使得自定义的端点暴露。②@EndPoint中的id命名不能使用驼峰法。

启动应用后，发送请求，可得图8-8

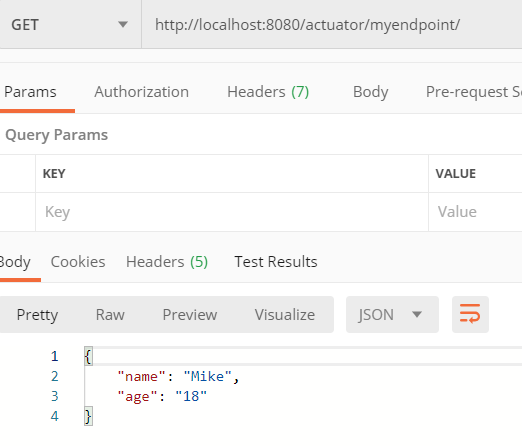


图8-8

@EndPoint：构建端点的id

@ReadOperation，@WriteOperation，@DeleteOperation分别对应get，post，delete三种请求方式。

@Selector：可以增加路径参数，比如请求为<http://localhost:8080/actuator/myendpoint/19>

则响应如下：

{

"name": "Mike",

"age": "19"

}

同时不能忘记添加@Component将类注入到spring容器中。此外，@Endpoint 注解的 Bean，

所有的方法都将通过JMX自动暴露，并且在Web应用程序中也将通过HTTP暴露，还可以使用 @JmxEndpoint 或 @WebEndpoint 编写特定技术的端点。这些端点仅限于各自的技术。例如，@WebEndpoint 仅通过 HTTP 暴露，而不是 JMX。

# Spring Security

## Spring Security简介

在开发项目时，安全是至关重要的一环，尽管它容易被忽视。目前市面上开源的安全框架不少，例如 Apache Shiro、Spring Security。在这里我们将讲解Spring Security，它是一个为基于Spring的企业应用系统提供声明式的安全访问控制解决方案的安全框架，其前身是Acegi Security，它提供了一组可以在Spring应用上下问中配置的Bean，充分利用了Spring IOC(Inversion of Control)容器和AOP（Aspect Oriented Programming）功能，为应用系统提供声明式的安全访问控制功能。

Spring Security 支持许多依赖于Servlet过滤器，过滤器会拦截进入请求，并在应用程序处理该请求之前进行一些安全处理。我们可以根据自身需求，使用适当的过滤器来保护字节的应用程序。应用的安全性主要包括用户认证（Authentication）和用户授权（Authorization）两个部分。用户认证是指验证某个用户是否为系统中的合法主体，简单来说就是用户能否访问该系统，一般会要求用户提供用户信息，诸如用户名和密码之类；用户授权是指验证某个用户是否具有权限执行某个操作，不同用户具有不同的权限，例如一个网站的管理员和普通游客的权限是不同的。

## 配置Spring Security（这部分我结合实例来写，所以省略了）

## Spring Security入门

### 框架原理

Spring security的基本原理图：

Username

Password

AuthenticationFilter

…

Basic

AuthenticationFilter

Client

REST

API

FilterSecurity

Interceptor

Exception

TranslationFilter

Spring Security框架在用户发送来请求后会经过一个过滤器链然后才会到达我们自己编写的API，当接收到请求后，它会到达第一个过滤器UsernamePassowordAuthenticationFilter，

这个过滤器的作用是判断请求是否带有username和paassword这两个参数，如果携带则进行拦截验证，如果没有就进入到第二个过滤器BasicAuthenticationFilter，这个过滤器会判断请求头中是否含有需要验证的信息，如果没有进入下一个拦截器，最后到ExceptionTranslationFilter拦截器，它会捕获FilterSecurityInterceptor抛出的异常，而FilterSecurityInterceptor会判断这个请求是否校验通过，权限是否通过。所有的请求都需要进行校验通过，若没通过则抛出异常，然后这个异常处理后会重定向到第一个过滤器。

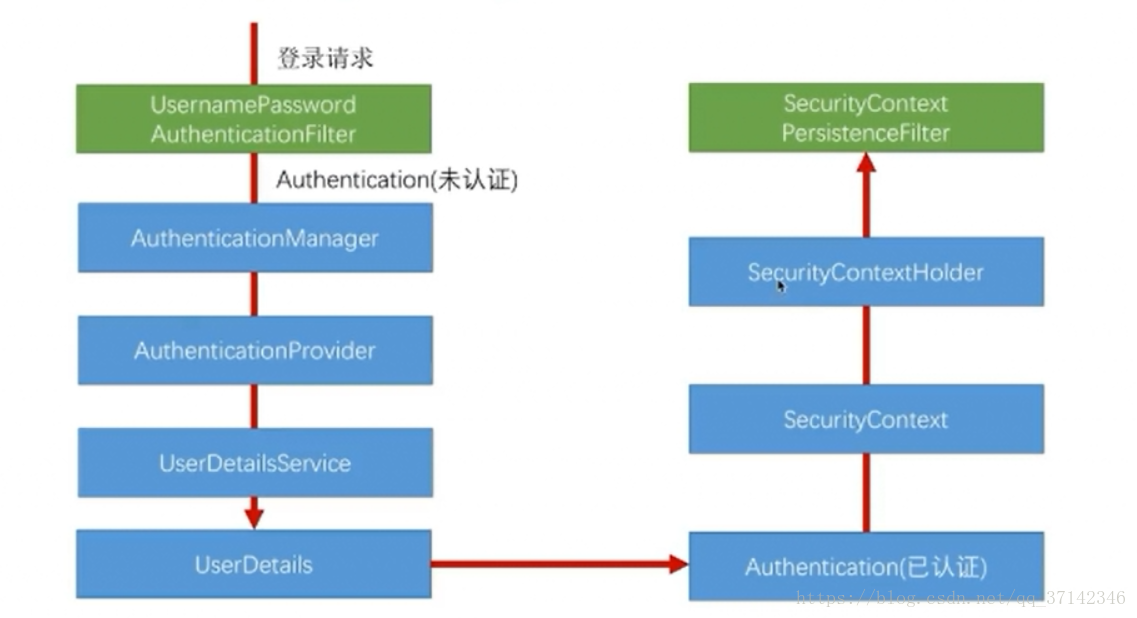


图9-1

认证流程如图9-1所示

### 核心组件

本节主要介绍Spring Security中的核心组件

1. SecurityContextHolder

SecurityContextHolder是最基本的组件类，它用于存储应用程序当前安全上下文的详细信息，比如一些用户的认证信息，拥有的权限。默认情况下，SecurityContextHolder使用ThreadLocal存储这些详细信息，在web环境下，Spring Secruity在用户登录时自动绑定认证信息到当前线程，在用户退出时，自动清除当前线程的认证信息。

1. 获取关于当前用户的信息

在SecurityContextHolder内部，我们存储了与应用程序交互的主体的详细信息。Spring Security使用Authentication对象来表示此信息。通常，无需自己创建Authentication对象，因为这些信息是与线程绑定的，所以可以在应用程序中的任何位置使用静态方法来获取当前经过身份验证的用户的名称，例如：

Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

**if** (principal **instanceof** UserDetails) {

String username = ((UserDetails)principal).getUsername();

} **else** {

String username = principal.toString();

}

调用getContext（）返回的对象是SecurityContext接口的一个实例。这是保存在线程本地存储中的对象。getAuthentication返回认证信息，getPrincipal()返回身份信息，UserDetails是Spring Security对用户信息的封装类。

3. Authentication

Authentication的源码如下：

package org.springframework.security.core;

import java.io.Serializable;

import java.security.Principal;

import java.util.Collection;

public interface Authentication extends Principal, Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

Object getCredentials();

Object getDetails();

Object getPrincipal();

boolean isAuthenticated();

void setAuthenticated(boolean var1) throws IllegalArgumentException;

}

Authentication继承自来自于java.security包下的Principal类，本身也是Spring.security包中的接口。

从这个接口中，我们可以得到如下的信息：

getAuthorities: 权限列表,通常是代表权限的字符串列表；

getCredentials: 密码信息,由用户输入的密码凭证，认证之后会移出，来保证安全性；

getDetails： 细节信息，Web应用中则是访问者的ip地址和sessionId；

getPrincipal： 身份信息，一般返回UserDetails的实现类；

4. UserDetailsService

public interface UserDetailsService {

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}

UserDetailsService接口上的唯一方法接受一个String基于用户名的参数，并返回UserDetails

5. UserDetails

UserDetails接口，它代表了最详细的用户信息, 这个接口涵盖了一些必要的用户信息字段，具体的实现类对它进行了扩展,源码如下：

public interface UserDetails extends Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}

这个接口类似于Authentication接口，包含了用户名，密码以及权限信息，不同的是Authentication中的getCredentials取到的是用户提交的密码凭证，而UserDetails中的getPassword取到的则是用户正确的密码信息，认证的第一步就是比较两者是否相同，除此之外，Authentication中的getAuthorities是认证用户名和密码成功之后，由UserDetails中的getAuthorities传递而来。而Authentication中的getDetails信息是经过了AuthenticationProvider认证之后填充的。

6. AuthenticationManager

AuthenticationManager接口包含一个方法，它的作用是认证，认证成功后会返回一个用户认证信息的实例。源码如下：

public interface AuthenticationManager {

Authentication authenticate(Authentication authentication)

throws AuthenticationException;

}

AuthenticationManager作为认证相关的核心接口，也是发起认证的出发点。实际业务中可能根据不同的信息进行认证。Spring Security中的推荐实现AuthenticationManager接口的ProviderManager来自定义自己的认证方式，它不处理身份验证请求本身，而是委派给已配置的AuthenticationProvider的列表，依次查询每个列表以查看其是否可以执行身份验证。

如果认证成功则立即返回，认证失败则返回null，继续下一个AuthenticationProvider认证。如果所有的AuthenticationProvider都无法认证成功，则ProviderManage会抛出一个

ProviderNotFoundException异常。AuthenticationManager源码如下：

public class ProviderManager implements AuthenticationManager, MessageSourceAware,

InitializingBean {

// 一个AuthenticationProvider列表

private List<AuthenticationProvider> providers = Collections.emptyList();

public Authentication authenticate(Authentication authentication)

throws AuthenticationException {

Class<? extends Authentication> toTest = authentication.getClass();

AuthenticationException lastException = null;

Authentication result = null;

// 依次认证

for (AuthenticationProvider provider : getProviders()) {

if (!provider.supports(toTest)) {

continue;

}

try {

result = provider.authenticate(authentication);

if (result != null) {

copyDetails(authentication, result);

break;

}

}

catch (AuthenticationException e) {

lastException = e;

}

}

// 如果有Authentication信息，则直接返回

if (result != null) {

if (eraseCredentialsAfterAuthentication

&& (result instanceof CredentialsContainer)) {

((CredentialsContainer) result).eraseCredentials();

}

//发布登录成功事件

eventPublisher.publishAuthenticationSuccess(result);

return result;

}

//若没有认证成功，包装异常信息

if (lastException == null) {

lastException = new ProviderNotFoundException(messages.getMessage(

"ProviderManager.providerNotFound",

new Object[] { toTest.getName() },

"No AuthenticationProvider found for {0}"));

}

prepareException(lastException, authentication);

throw lastException;

}

}

7. AuthenticationProvider

AuthenticationProvider的源码如下：它提供了两个方法，一个是认证方法；另一个是判断是否满足进行认证的条件。

public interface AuthenticationProvider {

Authentication authenticate(Authentication authentication)

throws AuthenticationException;

boolean supports(Class<?> authentication);

}

spring提供了一些AuthenticationProvider的实现，例如DaoAuthenticationProvider，AnonymousAuthenticationProvider等等；DaoAuthenticationProvider是从数据库中读取用户信息和前台提交的用户信息进行对比认证。

## 实例

1.简单使用

首先新建一个springboot项目，并导入spring security依赖

<!-- spring security依赖 -->

<dependency>

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

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

</dependency>

运行项目，然后在浏览器中输入localhost：8080，会看到一个登录页面，这个页面是Spring security默认的登录页面，如图9-2所示

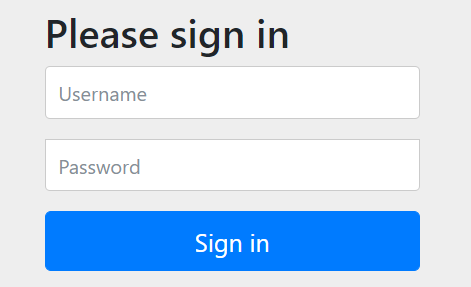
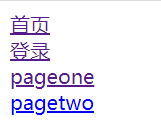


图9-2

这里有登录用户名和密码，security默认的用户名是user，默认密码是项目启动时，通过UUID算法随机生成，可以从控制台中看到默认密码。登录之后会显示登录，但是此时是一个默认的错误页面，因为项目中没有任何测试页面。

2.用户名密码认证和角色权限控制

（1）在项目中简单编写首页，登录页面，以及两个功能页面



首页如图



登录页如图

（2）新建controller包以及一个controller类

@Controller

public class TestConroller {

@RequestMapping({"/","/index"})

public String index(){

return "index";

}

@RequestMapping("toLogin")

public String toLogin(){

return "login";

}

@RequestMapping("/pageone")

public String pageone(){

return "pageone";

}

@RequestMapping("/pagetwo")

public String pagetwo(){

return "pagetwo";

}

}

实现页面的简单跳转。

（3）新建config包，在包中新建一个SecurityConfig类

这个类继承了WebSecurityConfigurerAdapter类，然后在其中可以重写配置方法。

@EnableWebSecurity

public class Securityconfig extends WebSecurityConfigurerAdapter {

//授权

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

表示首页所有人都可以访问

.antMatchers("/").permitAll()

.antMatchers("/pageone").hasRole("vip1")

pageone页面只有vip1权限可以访问

.antMatchers("/pagetwo").hasRole("vip2");

//没有授权默认会到登录页面,需要开启登录页面

http.formLogin().loginPage("/toLogin");

}

//认证

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication().passwordEncoder(new BCryptPasswordEncoder())

.withUser("user1").password(new BCryptPasswordEncoder().encode("1234")).roles("vip1")

开启了密码编码加密功能，其中有多种密码编码方式

.and()

.withUser("user2").password(new BCryptPasswordEncoder().encode("1234")).roles("vip1","vip2");

赋予用户user2的权限是vip1和vip2，意味着用户可以访问pageone和pagetwo两个页面

}

}

@EnableWebSecurity注解开启了Spring Security的功能

一般情况下，用户的信息都是从数据库中存取的，而这个例子里面的用户信息都是保存在内存中的。

用户信息与数据库的交互可以用以下方法：

@Autowired

private DataSource dataSource;

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

// ensure the passwords are encoded properly

UserBuilder users = User.withDefaultPasswordEncoder();

auth

.jdbcAuthentication()

.dataSource(dataSource)

.withDefaultSchema()

.withUser(users.username("user").password("password").roles("USER"))

.withUser(users.username("admin").password("password").roles("USER","ADMIN"));

}