# SpringBoot介绍

相关链接：

https://www.jianshu.com/p/b0b5cc08a6f4?utm\_campaign=hugo&utm\_medium=reader\_share&utm\_content=note&utm\_source=qq –未完成

## 核心特性

SpringBoot最重要的4大核心特性：自动配置、起步依赖、Actuator、命令行界面(CLI)。我们仅需要关注前面3个特性；CLI是SpringBoot的可选特性，功能强大的同时也引入了一套不常规的开发模型。

# 常见注解(Annotation)

## @SpringBootApplication注解

--用在Spring Boot 主类上，用来开启Spring Boot的各项能力。

它是@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan这三个注解的组合。

@EnableAutoConfiguration注解：

--允许自动配置注解，根据当前路径下的包或者类来配置Spring Bean

@Configuration注解：

## @Bean

任何标记了@Bean的方法，其返回值将作为一个bean注册到Spring的IOC容器中，方法名默认成为该Bean定义的id

## @Configuration

最初Spring使用xml配置文件来描述bean的定义及相关依赖关系，从而生成大量xml文件。

-1）xml描述bean的定义：

<bean id="bookService" class="com.yjj.BookService"></bean>

-2）注解描述

@Configuration

public class BookConfiguration {

@Bean

public BookService bookService() {

return new BookService();

}

}

存在依赖关系：

-1）xml描述

<bean id="bookService" class="com.yjj.BookService">

<property name="userService" ref="userService" />

</bean>

-2）注解描述

@Configuration

public class BookConfiguration {

@Bean

public BookService bookService(UserService userService) {

// return new BookService(userService()); //引入UserService的另一种方式（局限于userService()方法是本类的成员方法）

return new BookService(userService);

}

@Bean

public UserService userService() {

return new UserService();

}

}

注：多次调用userService()方法时，得到的UserService实例仅有一个。

## @ComponentScan

该注解对应xml配置的<comtext:component-scan>元素，表示启用组件扫描，Spring会自动扫描所有通过注解配置的bean，然后将其注册到ioc容器中。

可通过basePackages属性来指定扫描范围，默认从声明@Component所在类的package进行扫描。

## @Import

该注解用于导入另一个配置类。

可同时引入多个配置类，如：@Import({A.class, B.class})

例如：

@Import(UserConfiguration.class) //引入User配置类

@Configuration

public class BookConfiguration {

@Bean

public BookService bookService(UserService userService) {

return new BookService(userService);

}

}

@Configuration

public class UserConfiguration {

@Bean

public UserService userService() {

return new UserService();

}

}

## @Conditional

# SpringCache缓存

## 概述

Spring3.1引入了基于注解的缓存技术，它本质上不是一个具体的缓存实现方案（如EhCache、redis、mencached等），而是一个对缓存使用的抽象！

SpringCache提供了一个CacheManager接口，用于抽象和管理缓存；提供Cache接口来实现具体缓存方案（如EhCache、redis等）；而业务数据的CRUD操作，则由@Cacheable/@CachePut/@CacheEviet注解来进行配置后，由Cache接口下的各种实现类来处理。

SpringCache不提供失效时间相关配置：因为并不是所有的具体缓存实现都支持设置失效时间的。

传统缓存实现：

相关教程：https://www.cnblogs.com/zsychanpin/p/7191021.html

流程：先查询缓存数据，有则直接返回，无则从数据库获取。这样做的缺点是代码耦合性太高。

getUser(String username) {

User user = getUserByCache(username); //取缓存数据

if (user == null) {

user = getUserByDB(username); //取数据库数据

}

return user;

}

## 简单使用

-1）pom.xml引入SpringCache依赖：

<dependency>

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

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

</dependency>

-2）创建配置类

@Configuration

@EnableCaching

public class CachingConfig {

@Bean

public CacheManager cacheManager() {

SimpleCacheManager cacheManager = new SimpleCacheManager();

//配置名为accountCache的ConcurrentMapCache具体缓存方案(可配置多个)

cacheManager.setCaches(Arrays.asList(new ConcurrentMapCache("userCache")));

return cacheManager;

}

}

-3）测试

public class User {

private String name;

private Date birthday;

private Integer age;

// get/set方法

}

@Service

public class UserService {

public User findOne(String username) {

System.out.println("模拟从数据库取数据...");

return new User(username, Date.valueOf("1995-05-05"), 24);

}

public User addUser(String username) {

System.out.println("模拟将数据写入数据库...");

return new User(username, Date.valueOf("2018-01-01"), 20);

}

public User delUser(String username) {

System.out.println("模拟将数据从数据库删除...");

return new User(username, Date.valueOf("2018-01-01"), 20);

}

}

@RestController

public class UserController {

@Autowired

private UserService userService;

@Cacheable(value="userCache", key="'username='.concat(#root.args[0])")

@RequestMapping("/findUser")

public User findUser(String username) {

return userService.findOne(username);

}

@CachePut(value="userCache", key="'username='.concat(#root.args[0])")

@RequestMapping("/addUser")

public User addUser(String username) {

return userService.addUser(username);

}

@CacheEvict(value="userCache", key="'username='.concat(#root.args[0])")

@RequestMapping("/delUser")

public User delUser(String username) {

return userService.delUser(username);

}

}

-4）执行

http://localhost:8080:findUser?name=testAA //执行几次，发现控制台仅输出一行内容

说明：当访问该方法时，会从userCache缓存中查询是否存在key，存在则将key对应的value返回（不会打印内容），不存在则执行该方法（会打印内容）并将返回值存入缓存中。

实例二：

//先执行update方法，再清空缓存名userCache下key为username的数据

@CacheEvict(value="userCache", key="#user.getName()")

public void updateAccount(User user) { //...}

// 清空缓存名为userCache下的所有数据

@CacheEvict(value="userCache", allEntries=true)

public void reload() {}

测试：

//1.执行实际方法，再缓存数据（此时缓存有key=name1的数据）

Account account = accountService.getAccountByName("name1");

//2.先执行实际方法，再清空缓存（此时缓存无key为name1的数据）

accountService.updateAccount(account);

//3.执行执行实际方法，再缓存数据（此时缓存将key=name1的数据再次存入）

accountService.getAccountByName("name1");

实例三：

// 先执行更新方法，再将结果存入缓存中（不能用@Cacheable注解）

@CachePut(value="accountCache",key="#account.getName()")

public Account updateAccount(Account account) { //... }

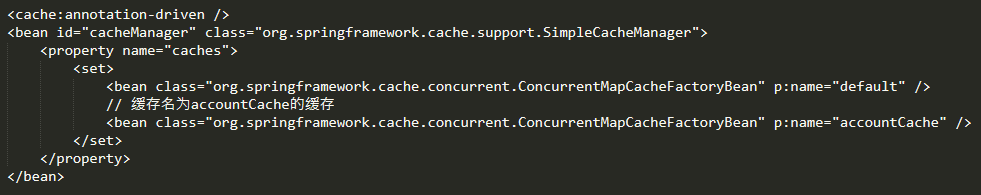
## 缓存管理器CacheManager

两种形式引入CacheManager：

-1）xml配置：需要加入<cache:annotation-driven />和CacheManager的Bean

-2）在配置类注解引入：需添加@EnableCaching和CacheManager的Bean

Xml配置形式如下：



<cache:annotation-driven/>：自动发现并配置CacheManager

缓存管理器有一个spring的缺省实现，即SimpleCacheManager类，它仅需配置caches属性，即此缓存管理器的引入的具体缓存实现方案。

该缓存管理器引入了两个具体的缓存：一个缺省名为default的本地缓存，一个名为accountCache的本地缓存。本地缓存使用的是ConcurrentMapCacheFactoryBean，它是基于java.util.concurrent.ConcurrentHashMap的一个内存缓存实现方案。

## 注解配置说明

### Spring的cache注解说明

--详情查看org.springframework.cache.annotation包。

前提：@EnableCaching --开启缓存（在启动类或配置类添加）。这样才能使用底下的注解

-1）@Cacheable

--缓存查询。先从缓存中查询有无数据，有则直接返回，无则执行方法并将返回值存入缓存。适用于读数据

主要属性：value/cacheName、key、condition/unless

value/cacheName：缓存名称，表明使用哪个具体缓存方案

key：该缓存下的唯一标识key，可使用SpEL语法

condition/unless：表明缓存条件，可以使用SpEL语法

-2）@CachePut

--缓存添加。先执行方法，后将返回值存入缓存。适用于写数据

主要属性：value/cacheName、key、condition/unless、beforeInvocation

beforeInvocation：false-在实际方法执行后清空缓存，若实际方法抛异常时不清空缓存，true-在实际方法执行前清空缓存

-3）@CacheEvict

--缓存清除。先执行方法，后通过key将缓存里的数据删除。适用于删数据

主要属性：value/cacheName、key、allEntries

allEntries：表明是否删除命名空间下的所有缓存，默认为false

-4）@Caching --构建复杂缓存。

### 自定义key生成

有两种形式的key配置：

-1）自定义keyGenerator

-2）手动配置（使用root对象）

a）可使用root对象生成key（#root对象可省略）：

methodName-当前方法名

method-当前方法

target-当前被调用的对象

targetClass-当前被调用的对象的class

args-当前方法参数组成的数组

caches-当前被调用的方法使用的Cache

例如：

@Cacheable(value = "sampleCache", key="targetClass.getName()+'.'+methodName+'.'+#id")

public String getBook(int id) { //... }

b）keyGenerator属性

Spring还提供了org.springframework.cache.interceptor.KeyGenerator接口,使用keyGenerator去指定实现此接口的bean的名字。

例如：

--1）xml引入

<bean id="cacheKeyGenerator" class="com.yjy.util.KeyGenerator" />

<cache:annotation-driven cache-manager="cacheManager" key-generator="cacheKeyGenerator" proxy-target-class="true" />

--2）使用

@Cacheable(cacheNames="sampleCache", keyGenerator="myKeyGenerator")

public String getBook(int id) { //... }