# Spring Boot概述

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。具有如下优点：

独立运行、简化配置、自动配置、无代码生成和XML配置、应用监控、上手容易。

**简化配置**

SpringBoot是对Spring的进一步封装，基于注解开发，舍弃笨重的XML，确实需要配置的使用yml或properties进行简要配置即可。Xml文件使用javaConfig代替，XML中bean的创建，使用@bean代替后可以直接注入。

**Spring Boot自动配置原理**

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

**产品级独立运行**

每一个工程都可以打成一个jar包，其中内置了Tomcat或其他Servlet容器，可以独立运行，这是和微服务理念最为契合的一点。

**强大的场景启动器**

每一个特定场景下的需求都封装成了一个starter，只要导入这个starter就有了这个场景所需要的一切。其中包括针对这个场景的自动化配置、依赖信息。

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

**组件**

starter依赖、@EnableXxx注解、yml或properties配置和主启动类。

# Spring Boot入门实例

## 创建工程

每次创建工程都必须联网；必须借助Spring插件。

### IDEA创建Spring Boot工程

New Module→

Spring Initializr（

Module SDK选择JDK版本；

Choose Initializr Service URL中选择Default即可）→

Project Metadata（

Type选择创建的工程类型；

Packaging选择打包方式，一般选择jar方式打包）→

Dependencies（

Web中选择Spring Web Starter的依赖）

### Maven依赖配置文件

通过继承spring-boot-starter-parent项目或导入spring-boot-dependencies项目依赖来开启Spring Boot特性。

----pom.xml-----------------------------------------------------------------------------------

<!-- 继承SpringBoot官方指定的父工程 -->

<parent>

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

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

<version>2.1.6.RELEASE</version>

</parent>

<dependencies>

<!-- 加入Web开发所需要的场景启动器 -->

<dependency>

<!-- 指定groupId和artifactId即可，版本已在父工程中定义 -->

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

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

</dependency>

</dependencies>

### 主启动类

要在Spring Boot启动时运行特定代码，可实现接口ApplicationRunner或者 CommandLineRunner，这两个接口实现方式一样，都只提供了一个 run 方法。

----SpringboothelloApplication.java-----------------------------------------------------------

// SpringBoot默认包扫描规则：“主启动类所在包”和“主启动类所在包的子包”会被自动扫描到。

// 如果无法满足默认规则，那么可使用@ComponentScan注解明确指定要扫描的包，此时默认规则会失效。

// @ComponentScan("com.example.spring.bo2ot.handler")

@SpringBootApplication

public class SpringboothelloApplication {

public static void main(String[] args) {

SpringApplication.run(SpringboothelloApplication.class, args);

}

}

### Handler类

----HelloHandler.java-------------------------------------------------------------------------

@RestController

public class HelloHandler {

@RequestMapping("/say/hello/spring/boot")

public String sayHelloToSpringBoot() {

return "Hello SpringBoot!";

}

}

## 运行Spring Boot工程

### 运行方式

1）打包用命令或者放到容器中运行

2）用 Maven/ Gradle 插件运行

3）直接执行 main 方法运行

### 运行原理

具体功能注解 pom.xml

↓ ↓

autoconfigure starter-xxx

↓ ↓

查找对应的class → xxx.jar

**XxxAutoConfiguration**

这样的类是用来做自动配置，SpringBoot中存在大量这样的类。每一个自动配置类都将这个场景所需要的所有组件都注册到容器中，并按照普遍接受的通用配置方式配置好。而这些默认配置通常都是行业中多年积累的最佳实践。当然，如果我们不满意默认配置，也可以根据我们的需要在yml或properties文件中重新配置。

在spring-boot-autoconfigure.jar中的META-INF/spring.factories文件中定义了大量自动配置类。这些类都可以在这个jar包中找到。

# Spring Boot配置文件

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。springboot会优先读取bootstrapp.yaml配置文件，bootstrap 配置文件有以下几个应用场景：使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；一些固定的不能被覆盖的属性；一些加密/解密的场景。

SpringBoot环境下常用的配置文件有两种，一种是properties属性文件，一种是yml文件。二者各有特点，语法也有很大区别，但是最终效果基本一致。不管使用哪一种都必须遵守SpringBoot配置文件的规则。

**文件名**

不管扩展名使用yml还是properties，文件名都必须是application。也就是说，有效的文件名只有下面二者之一：application.yml或application.properties。

**存放目录**

Maven工程**：**src/main/resouces目录

普通Java工程：类路径/config目录

## properties配置文件

配置文件名为application.properties，语法格式为xxx.xxx.xxx=xxx

## yml配置文件

yml是YAML（YAML Ain't Markup Language）语言的文件，以数据为中心，比json、xml等更适合做配置文件。语法规范请参考：http://www.yaml.org。

**语法**

使用缩进表示层级关系；

缩进时不允许使用Tab键，只允许使用空格；

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可；

大小写敏感。

**数据结构**

对象：键值对的集合

数组：一组按次序排列的值

字面量：单个的、不可再分的值

**对象**

对象是一系列嵌套的键值对，键和值之间用冒号分隔，但要求冒号后面必须有空格。

----application.yml---------------------------------------------------------------------------

spring:

application:

name: apple

server:

port: 8181

context-path: /banana

行内写法

----application.yml---------------------------------------------------------------------------

student: {stuId: 100, stuName: tom, stuAge: 20}

**数组**

数组是用“- ”开头的一组值。

----application.yml---------------------------------------------------------------------------

fruit:

- apple

- banana

- orange

行内写法

----application.yml---------------------------------------------------------------------------

fruit: [apple, banana, orange]

## 获取配置文件中的值

读取配置文件数据的类必须也在容器中，所以使用@Component注解（当然要注意所在包必须能够被扫描），

# Spring Boot注解

## @SpringBootApplication

该注解是Spring Boot的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

## @PropertySource

读取配置

## @Value

读取配置

可以使用@Value注解读取简单值，@Value注解的属性值是${}括起来的表达式。

----Java--------------------------------------------------------------------------------------

@Value("${spring.application.name}")

private String appName;

----yml---------------------------------------------------------------------------------------

spring:

application:

name: aaa

## @Environment

读取配置

## @ConfigurationProperties

来读取配置

使用@ConfigurationProperties表示从配置文件读取信息，prefix指定开始读取信息的位置。

----Student.java------------------------------------------------------------------------------

@Component

@ConfigurationProperties(prefix="sb")

public class Student {

private Integer stuId;

private String stuName;

private List<String> subjectList;

private Date birthday;

private Map<String, String> stuMap;

}

----application.yml---------------------------------------------------------------------------

sb:

stu-id: 23 #stu-id和Student类中的stuId对应，这两种命名规范是对应的

stu-name: tom

birthday: 1990/10/15

subject-list:

- java

- php

- mysql

stuMap:

k1: v1

k2: v2

引入spring-boot-configuration-processor将能够在编写yml的时候有提示

----pom.xml-----------------------------------------------------------------------------------

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

**重要注解**

| **注解名称** | **作用** |
| --- | --- |
| @SpringBootApplication | 声明一个SpringBoot程序，并使 @SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan 等注解生效 |
| @SpringBootConfiguration | 相当于@Configuration注解的重新定义 |
| @Configuration | 声明一个Spring配置类 |
| @Bean | 在@Configuration注解标记的类中将标记了@Bean的方法返回值对象加入IOC容器，可以对应XML配置文件中的bean标签来理解 |
| @EnableAutoConfiguration | 启用自动配置 |
| @AutoConfigurationPackage | 定义默认的包扫描规则 |
| @ComponentScan | 指定要扫描的包 |
| @Import | 导入所有的自动配置场景 |

# Spring Boot分布式事务

第一种方案：

可靠消息最终一致性，需要业务系统结合MQ消息中间件实现，在实现过程中需要保证消息的成功发送及成功消费。即需要通过业务系统控制MQ的消息状态。

第二种方案：

TCC补偿性，分为三个阶段TRYING-CONFIRMING-CANCELING。每个阶段做不同的处理。

TRYING阶段主要是对业务系统进行检测及资源预留；

CONFIRMING阶段是做业务提交，通过TRYING阶段执行成功后，再执行该阶段。默认如果TRYING阶段执行成功，CONFIRMING就一定能成功。

CANCELING阶段是回对业务做回滚，在TRYING阶段中，如果存在分支事务TRYING失败，则需要调用CANCELING将已预留的资源进行释放。

# Spring Boot集成持久层框架

## 集成Mybatis

## 集成Mybatis-Plus

## 整合数据库

Spring Boot可以和持久层框架MyBatis或MyBatis Plus进行整合，整合后需要在配置文件中配置数据库驱动、URL、数据库用户名和数据库密码等参数。

SpringBoot整合MySQL数据库时根据MySQL版本不同，其配置文件的数据库驱动配置和URL配置也有所变化，整合MySQL5时配置文件如下：

|  |
| --- |
| spring:  datasource:  url: jdbc:mysql://localhost:3306/test?useSSl=false  driver-class-name: com.mysql.jdbc.Driver |

整合MySQL8以上时配置文件如下，其中URL中需要添加serverTimeZone参数且该参数的值为GMT2B8，因为Spring Boot 2集成了MySQL8版本的JDBC驱动，这个版本的驱动需要添加这个参数，否则连接数据库时会抛出如下异常

java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more

且MySQL8对应的JDBC驱动建议使用com.mysql.cj.jdbc.Driver，之前的com.mysql.jdbc.Driver驱动已经被废弃，如果使用旧的驱动就会有警告信息。

|  |
| --- |
| spring:  datasource:  url: jdbc:mysql://localhost:3306/test?serverTimezone=GMT%2B8  driver-class-name: com.mysql.cj.jdbc.Driver |

首先导入持久化层的依赖

----pom.xml-----------------------------------------------------------------------------------

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.12</version>

</dependency>

<dependency>

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

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

<version>1.1.1</version>

</dependency>

然后编写Mapper相关的内容

----Emp.java----------------------------------------------------------------------------------

public class Emp {

private Integer empId;

private String empName;

private Integer empAge;

...

}

----table\_emp---------------------------------------------------------------------------------

create table `table\_emp` (

`emp\_id` int not null auto\_increment ,

`emp\_name` varchar(100) null ,

`emp\_age` int null ,

primary key (`emp\_id`)

)

----EmpMapper.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.atguigu.spring.boot.mapper.EmpMapper">

<select id="selectAll" resultType="com.atguigu.spring.boot.entity.Emp">

select emp\_id empId, emp\_name empName, emp\_age empAge

from table\_emp

</select>

</mapper>

----EmpMapper.java----------------------------------------------------------------------------

public interface EmpMapper {

List<Emp> selectAll();

}

----EmpServiceImpl.java-----------------------------------------------------------------------

@Service

@Transactional

public class EmpServiceImpl implements EmpService {

@Autowired

private EmpMapper empMapper;

@Override

public List<Emp> getAll() {

return empMapper.selectAll();

}

}

----EmpHandler.java---------------------------------------------------------------------------

@Autowired

private EmpService empService;

@ResponseBody

@RequestMapping("/getAll")

public List<Emp> getAll() {

return empService.getAll();

}

----application.yml---------------------------------------------------------------------------

spring:

datasource:

name: mydb

type: com.alibaba.druid.pool.DruidDataSource

url: jdbc:mysql://127.0.0.1:3306/sb\_db

username: root

password: root

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

mybatis:

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

logging:

level:

com.example.springboothello.mapper: debug

在SpringBoot环境下打印日志，使用logging.level.pattern属性指定日志级别。pattern使用需要打印日志的包的名称，可以使用\*号代表多个包。

可以在主启动类上使用注解配置需要扫描Mapper

----SpringBootApplication.java----------------------------------------------------------------

// 使用@MapperScan注解指定Mapper扫描的包

@MapperScan("com.example.springboothello.mapper")

@SpringBootApplication

public class SpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(SpringboothelloApplication.class, args);

}

}

# SpringBoot集成缓存

SpringBoot通过在pom.xml配置文件中如下配置来集成缓存，内部实现了对Lettuce和jedis两个客户端的封装，默认使用的是Lettuce。使用缓存时只需要知道springboot只需要使用springboot提供的API，而不用了解具体的客户端的API。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId> </dependency> |

想要使用jedis作为操作redis的客户端，可以进行如下配置：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  <!-- 排除lettuce包，使用jedis代替-->  <exclusions>  <exclusion>  <groupId>io.lettuce</groupId>  <artifactId>lettuce-core</artifactId>  </exclusion>  </exclusions> </dependency> <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId> </dependency> |

此时通过调用获取工厂类方法可以获取到Jedis客户端工厂类

org.springframework.data.redis.connection.jedis.JedisConnectionFactory。

|  |
| --- |
| public void getConnectionFactory(){  *log*.info("redis连接工厂：{}",redisTemplate.getConnectionFactory()); } |

## 缓存相关类继承关系

**Object**

**|-- RedisAccessor**

**|-- RedisTemplate**

**|-- StringRedisTemplate**

## 缓存相关API

springboot默认注入了RedisTemplate和StringRedisTemplate模板类， StringRedisTemplate继承了RedisTemplate<String, String>类（此后均已StringRedisTemplate为例）。StringRedisTemplate只能对key=String，value=String的键值对进行操作，RedisTemplate可以对任何类型的key-value键值对操作。当redis存取的数据是字符串类型数据时，使用StringRedisTemplate，如果数据是复杂的对象类型，而取出的时候又不想做任何的数据转换，直接从redis里面取出一个对象，那么使用RedisTemplate。

他们各自序列化的方式不同，但最终都是得到了一个字节数组，StringRedisTemplate使用的是StringRedisSerializer类；RedisTemplate使用的是JdkSerializationRedisSerializer类。反序列化，则是一个得到String，一个得到Object

### Redistemplate接口方法

|  |
| --- |
| **RedisTemplate接口方法摘要** |
| public <T> T execute(RedisCallback<T> action)  public <T> T execute(RedisCallback<T> action, boolean exposeConnection)  public <T> T execute(RedisCallback<T> action, boolean exposeConnection, boolean pipeline)  public <T> T execute(SessionCallback<T> session)  public <T> T execute(RedisScript<T> script, List<K> keys, Object... args)  public <T> T execute(RedisScript<T> script, RedisSerializer<?> argsSerializer, RedisSerializer<T> resultSerializer,  List<K> keys, Object... args) |
| public List<Object> executePipelined(SessionCallback<?> session)  public List<Object> executePipelined(SessionCallback<?> session, @Nullable RedisSerializer<?> resultSerializer)  public List<Object> executePipelined(RedisCallback<?> action)  public List<Object> executePipelined(RedisCallback<?> action, @Nullable RedisSerializer<?> resultSerializer) |
| public <T extends Closeable> T executeWithStickyConnection(RedisCallback<T> callback) |
| private Object executeSession(SessionCallback<?> session) |
| public Boolean delete(K key)  public Long delete(Collection<K> keys) |
| public Boolean unlink(K key)  public Long unlink(Collection<K> keys) |
| public Boolean hasKey(K key) |
| public Long countExistingKeys(Collection<K> keys) |
| public Boolean expire(K key, final long timeout, final TimeUnit unit) |
| public Boolean expireAt(K key, final Date date) |
| public void convertAndSend(String channel, Object message) |
| public Long getExpire(K key)  public Long getExpire(K key, final TimeUnit timeUnit) |
| public Set<K> keys(K pattern) |
| public Boolean persist(K key) |
| public Boolean move(K key, final int dbIndex) |
| public K randomKey() |
| public void rename(K oldKey, K newKey) |
| ublic Boolean renameIfAbsent(K oldKey, K newKey) |
| public DataType type(K key) |
| public byte[] dump(K key) |
| public void restore(K key, final byte[] value, long timeToLive, TimeUnit unit, boolean replace) |
| public void multi() |
| public void discard() |
| public void watch(K key)  public void watch(Collection<K> keys) |
| public void unwatch() |
| public List<V> sort(SortQuery<K> query)  public <T> List<T> sort(SortQuery<K> query, @Nullable RedisSerializer<T> resultSerializer)  public <T> List<T> sort(SortQuery<K> query, BulkMapper<T, V> bulkMapper)  public <T, S> List<T> sort(SortQuery<K> query, BulkMapper<T, S> bulkMapper,  @Nullable RedisSerializer<S> resultSerializer)  public Long sort(SortQuery<K> query, K storeKey) |
| public ValueOperations<K, V> opsForValue()  public ListOperations<K, V> opsForList()  public <HK, HV> HashOperations<K, HK, HV> opsForHash()  public SetOperations<K, V> opsForSet()  public ZSetOperations<K, V> opsForZSet()  *返回操作五种类型数据结构的operations对象* |
| public BoundValueOperations<K, V> boundValueOps(K key)  public BoundListOperations<K, V> boundListOps(K key)  public <HK, HV> BoundHashOperations<K, HK, HV> boundHashOps(K key)  public BoundSetOperations<K, V> boundSetOps(K key)  public BoundZSetOperations<K, V> boundZSetOps(K key)  *返回操作五种类型数据结构的指定key的operations对象* |

### String数据结构

|  |
| --- |
| **ValueOperations接口方法摘要** |
| public V get(Object key)  *获取指定key的值* |
| V getAndSet(K key, V value);  *重新设置指定key的值并返回旧的值* |
| List<V> multiGet(Collection<K> keys);  *获取keys的值，并以list的形式返回* |
| String get(K key, long start, long end) |
| void set(K key, V value)  *设置指定key的值value*  void set(K key, V value, long timeout, TimeUnit unit)  default void set(K key, V value, Duration timeout)  *设置指定key的值value，并设置过期时间*  void set(K key, V value, long offset) |
| Boolean setIfAbsent(K key, V value);  *如果指定key不存在，则设置指定key的值value*  Boolean setIfAbsent(K key, V value, long timeout, TimeUnit unit)  *如果指定key不存在，则设置指定key的值value，并设置过期时间*  default Boolean setIfAbsent(K key, V value, Duration timeout)  *如果指定key不存在，则设置指定key的值value，并设置过期时间* |
| Boolean setIfPresent(K key, V value);  Boolean setIfPresent(K key, V value, long timeout, TimeUnit unit)  default Boolean setIfPresent(K key, V value, Duration timeout) |
| void multiSet(Map<? extends K, ? extends V> map) |
| Boolean multiSetIfAbsent(Map<? extends K, ? extends V> map) |
| Long increment(K key)  Long increment(K key, long delta)  Double increment(K key, double delta) |
| Long decrement(K key);  Long decrement(K key, long delta) |
| Integer append(K key, String value) |
| Long size(K key) |
| Boolean getBit(K key, long offset) |
| List<Long> bitField(K key, BitFieldSubCommands subCommands) |

### List数据结构

|  |
| --- |
| **ListOperations接口方法摘要** |
| List<V> range(K key, long start, long end); |
| void trim(K key, long start, long end); |
| Long size(K key); |
| Long leftPush(K key, V value);  Long leftPush(K key, V pivot, V value); |
| Long leftPushAll(K key, V... values);  Long leftPushAll(K key, Collection<V> values); |
| Long leftPushIfPresent(K key, V value); |
| Long rightPush(K key, V value);  Long rightPush(K key, V pivot, V value); |
| Long rightPushAll(K key, V... values);  Long rightPushAll(K key, Collection<V> values); |
| Long rightPushIfPresent(K key, V value); |
| void set(K key, long index, V value); |
| Long remove(K key, long count, Object value); |
| V index(K key, long index); |
| V leftPop(K key);  V leftPop(K key, long timeout, TimeUnit unit); |
| V rightPop(K key);  V rightPop(K key, long timeout, TimeUnit unit); |
| V rightPopAndLeftPush(K sourceKey, K destinationKey);  V rightPopAndLeftPush(K sourceKey, K destinationKey, long timeout, TimeUnit unit); |

### Hash数据结构

|  |
| --- |
| **HashOperations接口方法摘要** |
| Long delete(H key, Object... hashKeys); |
| Boolean hasKey(H key, Object hashKey); |
| HV get(H key, Object hashKey); |
| List<HV> multiGet(H key, Collection<HK> hashKeys); |
| Long increment(H key, HK hashKey, long delta);  Double increment(H key, HK hashKey, double delta); |
| Set<HK> keys(H key); |
| Long lengthOfValue(H key, HK hashKey); |
| Long size(H key); |
| void putAll(H key, Map<? extends HK, ? extends HV> m); |
| void put(H key, HK hashKey, HV value); |
| Boolean putIfAbsent(H key, HK hashKey, HV value); |
| List<HV> values(H key); |
| Map<HK, HV> entries(H key); |
| Cursor<Map.Entry<HK, HV>> scan(H key, ScanOptions options); |

### Set数据结构

|  |
| --- |
| **SetOperations接口方法摘要** |
| Long add(K key, V... values); |
| Long remove(K key, Object... values); |
| V pop(K key);  List<V> pop(K key, long count); |
| Boolean move(K key, V value, K destKey); |
| Long size(K key); |
| Boolean isMember(K key, Object o); |
| Set<V> intersect(K key, K otherKey);  Set<V> intersect(K key, Collection<K> otherKeys);  Set<V> intersect(Collection<K> keys); |
| Long intersectAndStore(K key, K otherKey, K destKey);  Long intersectAndStore(K key, Collection<K> otherKeys, K destKey);  Long intersectAndStore(Collection<K> keys, K destKey); |
| Set<V> union(K key, K otherKey);  Set<V> union(K key, Collection<K> otherKeys);  Set<V> union(Collection<K> keys); |
| Long unionAndStore(K key, K otherKey, K destKey);  Long unionAndStore(K key, Collection<K> otherKeys, K destKey);  Long unionAndStore(Collection<K> keys, K destKey); |
| Set<V> difference(K key, K otherKey);  Set<V> difference(K key, Collection<K> otherKeys);  Set<V> difference(Collection<K> keys); |
| Long differenceAndStore(K key, K otherKey, K destKey);  Long differenceAndStore(K key, Collection<K> otherKeys, K destKey);  Long differenceAndStore(Collection<K> keys, K destKey); |
| Set<V> members(K key); |
| V randomMember(K key); |
| Set<V> distinctRandomMembers(K key, long count); |
| List<V> randomMembers(K key, long count); |
| Cursor<V> scan(K key, ScanOptions options); |

### ZSet数据结构

|  |
| --- |
| **ZsetOperations接口方法摘要** |
| Boolean add(K key, V value, double score);  Long add(K key, Set<TypedTuple<V>> tuples); |
| Long remove(K key, Object... values); |
| Double incrementScore(K key, V value, double delta); |
| Long rank(K key, Object o); |
| Long reverseRank(K key, Object o); |
| Set<V> range(K key, long start, long end); |
| Set<TypedTuple<V>> rangeWithScores(K key, long start, long end); |
| Set<V> rangeByScore(K key, double min, double max);  Set<V> rangeByScore(K key, double min, double max, long offset, long count); |
| Set<TypedTuple<V>> rangeByScoreWithScores(K key, double min, double max);  Set<TypedTuple<V>> rangeByScoreWithScores(K key, double min, double max, long offset, long count); |
| Set<V> reverseRange(K key, long start, long end); |
| Set<TypedTuple<V>> reverseRangeWithScores(K key, long start, long end); |
| Set<V> reverseRangeByScore(K key, double min, double max);  Set<V> reverseRangeByScore(K key, double min, double max, long offset, long count); |
| Set<TypedTuple<V>> reverseRangeByScoreWithScores(K key, double min, double max);  Set<TypedTuple<V>> reverseRangeByScoreWithScores(K key, double min, double max, long offset, long count); |
| Long count(K key, double min, double max); |
| Long size(K key); |
| Long zCard(K key); |
| Double score(K key, Object o); |
| Long removeRange(K key, long start, long end); |
| Long removeRangeByScore(K key, double min, double max); |
| Long unionAndStore(K key, K otherKey, K destKey);  Long unionAndStore(K key, Collection<K> otherKeys, K destKey);  default Long unionAndStore(K key, Collection<K> otherKeys, K destKey, Aggregate aggregate);  Long unionAndStore(K key, Collection<K> otherKeys, K destKey, Aggregate aggregate, Weights weights); |
| Long intersectAndStore(K key, K otherKey, K destKey);  Long intersectAndStore(K key, Collection<K> otherKeys, K destKey);  default Long intersectAndStore(K key, Collection<K> otherKeys, K destKey, Aggregate aggregate);  Long intersectAndStore(K key, Collection<K> otherKeys, K destKey, Aggregate aggregate, Weights weights); |
| Cursor<TypedTuple<V>> scan(K key, ScanOptions options); |
| Set<V> rangeByLex(K key, Range range);  Set<V> rangeByLex(K key, Range range, Limit limit); |

# SpringBoot集成日志

## Slf4j

## logback

## log4j2