



Spring Boot

开发实用篇



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌



目录

Contents

- ◆ 热部署
- ◆ 配置高级
- ◆ 测试
- ◆ 数据层解决方案
- ◆ 整合第三方技术
- ◆ 监控



热部署

- 手动启动热部署
- 自动启动热部署
- 热部署范围配置
- 关闭热部署

启动热部署

- 开启开发者工具

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
```

- 激活热部署: **Ctrl + F9**

启动热部署

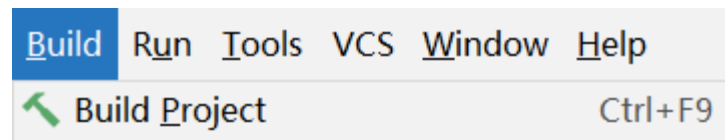
- 关于热部署
 - ◆ 重启 (Restart) : 自定义开发代码, 包含类、页面、配置文件等, 加载位置restart类加载器
 - ◆ 重载 (ReLoad) : jar包, 加载位置base类加载器



小结

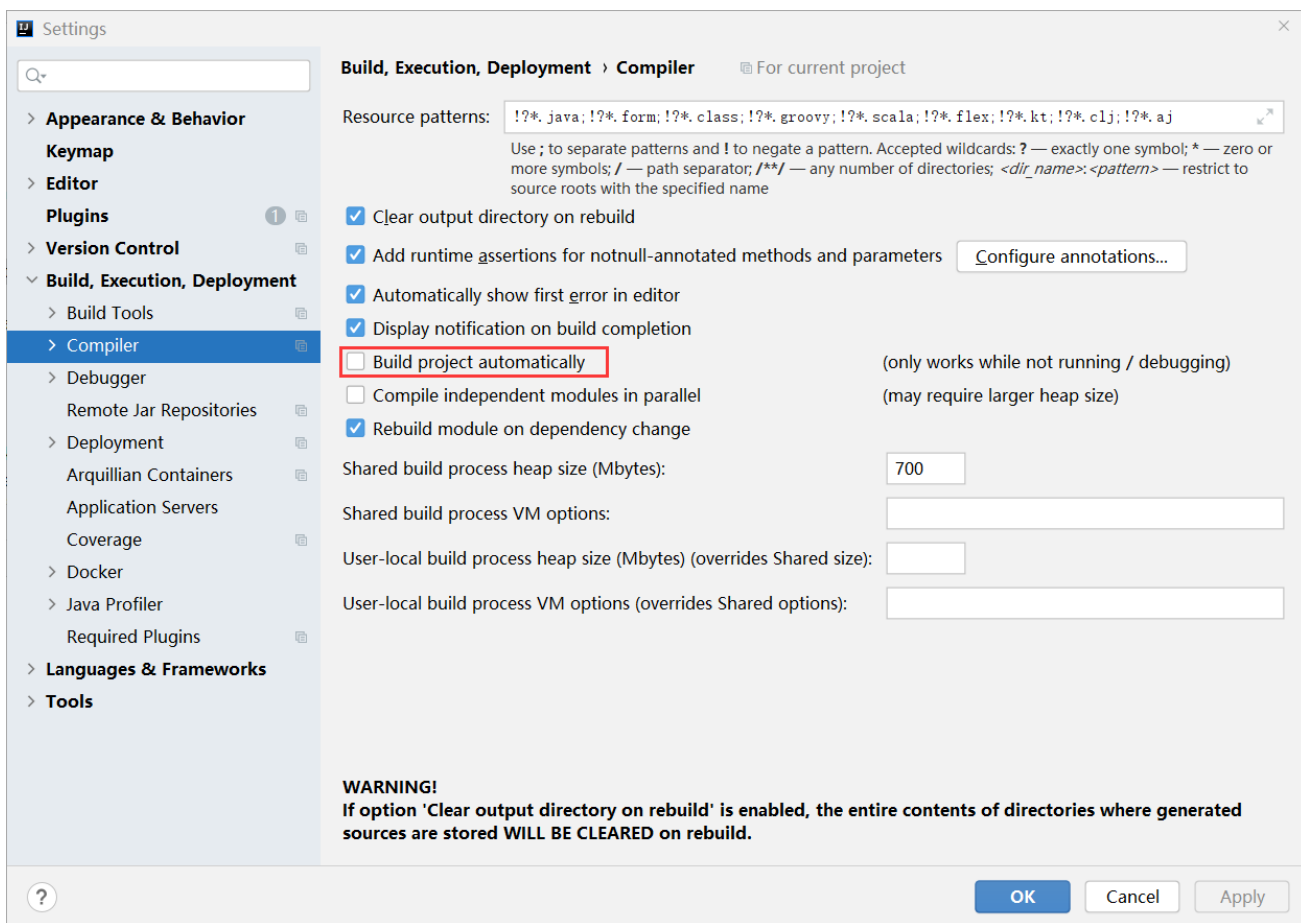
1. 开启开发者工具后启用热部署
2. 使用构建项目操作启动热部署 (Ctrl+F9)
3. 热部署仅仅加载当前开发者自定义开发的资源，不加载jar资源

启动热部署



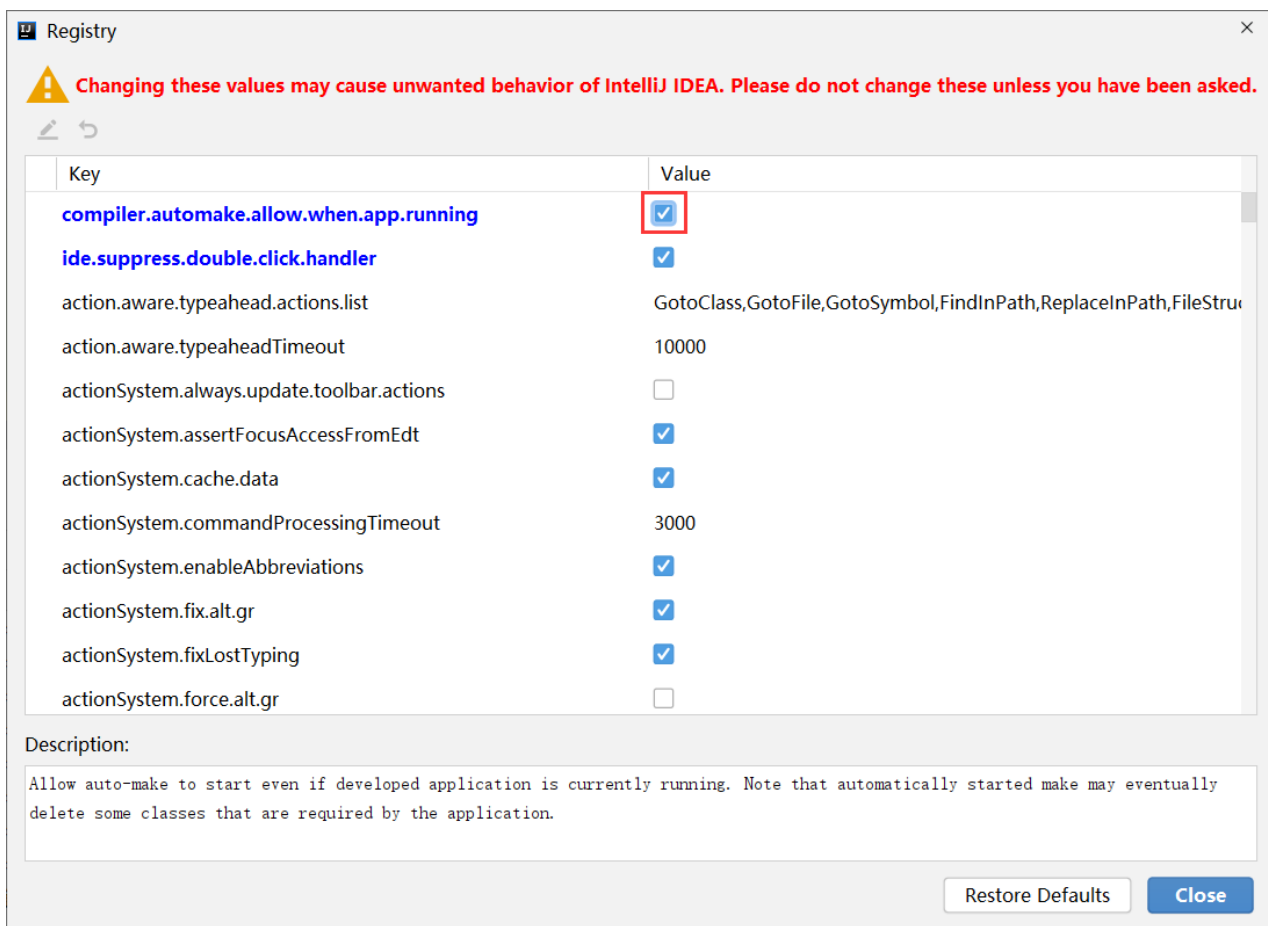
自动启动热部署

- 设置自动构建项目



自动启动热部署

- 设置自动构建项目



自动启动热部署

- 激活方式：Idea失去焦点5秒后启动热部署



小结

1. 设置自动构建用于自动热部署

热部署范围配置

- 默认不触发重启的目录列表
 - ◆ /META-INF/maven
 - ◆ /META-INF/resources
 - ◆ /resources
 - ◆ /static
 - ◆ /public
 - ◆ /templates

热部署范围配置

- 自定义不参与重启排除项

```
devtools:  
  restart:  
    exclude: public/**,static/**
```



小结

1. 自定义重启排除项

属性加载优先顺序

1. 参看<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config>

1. Default properties (specified by setting `SpringApplication.setDefaultProperties`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*` .
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env` .
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the test annotations for testing a particular slice of your application.
13. `@TestPropertySource` annotations on your tests.
14. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

低



高

禁用热部署

- 设置高优先级属性禁用热部署

```
public static void main(String[] args) {  
    System.setProperty("spring.devtools.restart.enabled", "false");  
    SpringApplication.run(SSMPApplication.class);  
}
```




小结

1. 禁用热部署功能



总结

1. 手动启动热部署
2. 自动启动热部署
3. 热部署范围配置
4. 关闭热部署



配置高级

- `@ConfigurationProperties`
- 宽松绑定/松散绑定
- 常用计量单位绑定
- 数据校验

@ConfigurationProperties

- 使用@ConfigurationProperties为第三方bean绑定属性

```
@Bean
@ConfigurationProperties(prefix = "datasource")
public DruidDataSource dataSource(){
    DruidDataSource ds = new DruidDataSource();
    return ds;
}
```

```
datasource:
  driverClassName: com.mysql.jdbc.Driver
```

@EnableConfigurationProperties

- @EnableConfigurationProperties注解可以将使用@ConfigurationProperties注解对应的类加入Spring容器

```
@SpringBootApplication
@EnableConfigurationProperties(ServerConfig.class)
public class DemoApplication {
}
```



```
//@Component
@Data
@ConfigurationProperties(prefix = "servers")
public class ServerConfig {
}
```

注意事项

@EnableConfigurationProperties与@Component不能同时使用

@ConfigurationProperties

- 解除使用@ConfigurationProperties注释警告

 Spring Boot Configuration Annotation Processor not configured [Open Documentation...](#) 

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
</dependency>
```



小结

1. `@ConfigurationProperties`可以为第三方bean绑定属性

宽松绑定

- @ConfigurationProperties绑定属性支持属性名宽松绑定

```
public class ServerConfig {  
    private String ipAddress;  
    private int port;  
    private long timeout;  
}
```

◆ 驼峰模式

```
servers:  
  ipAddress: 192.168.1.1  
  port: 2345  
  timeout: -1
```

◆ 下划线模式

```
servers:  
  ip_address: 192.168.1.2  
  port: 2345  
  timeout: -1
```

◆ 中划线模式

```
servers:  
  ip-address: 192.168.1.3  
  port: 2345  
  timeout: -1
```

◆ 常量模式

```
servers:  
  IP_ADDRESS: 192.168.1.4  
  port: 2345  
  timeout: -1
```


宽松绑定

- @ConfigurationProperties绑定属性支持属性名宽松绑定

```
public class ServerConfig {  
    private String ipAddress;  
    private int port;  
    private long timeout;  
}
```

◆ 驼峰模式

```
servers:  
  ipAddress: 192.168.1.1  
  port: 2345  
  timeout: -1
```

注意事项

宽松绑定不支持注解@Value引用单个属性的方式

宽松绑定

- @ConfigurationProperties绑定属性支持属性名宽松绑定

```
@Bean
@ConfigurationProperties(prefix = "datasource")
public DruidDataSource dataSource(){
    DruidDataSource ds = new DruidDataSource();
    return ds;
}
```

注意事项

绑定前缀名命名规范：仅能使用纯小写字母、数字、下划线作为合法的字符



小结

1. @ConfigurationProperties绑定属性支持属性名宽松绑定
2. @Value注解不支持松散绑定
3. 绑定前缀命名命名规则

常用计量单位

















- SpringBoot支持JDK8提供的时间与空间计量单位



```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
public class ServerConfig {
    private String ipAddress;
    private int port;
    private long timeout;
    @DurationUnit(ChronoUnit.MINUTES)
    private Duration serverTimeOut;
    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize dataSize;
}
```

常用计量单位






- JDK8支持的时间与空间计量单位



ChronoUnit

 DAYS	ChronoUnit
 SECONDS	ChronoUnit
 MINUTES	ChronoUnit
 HALF_DAYS	ChronoUnit
 CENTURIES	ChronoUnit
 DECADES	ChronoUnit
 ERAS	ChronoUnit
 FOREVER	ChronoUnit
 HOURS	ChronoUnit
 MICROS	ChronoUnit
 MILLENNIA	ChronoUnit
 MILLIS	ChronoUnit
 MONTHS	ChronoUnit
 NANOS	ChronoUnit
 WEEKS	ChronoUnit
 YEARS	ChronoUnit

Press Ctrl+. to choose the selected item afterwards [Next Tip](#)  

DataUnit

 MEGABYTES	DataUnit
 BYTES	DataUnit
 GIGABYTES	DataUnit
 KILOBYTES	DataUnit
 TERABYTES	DataUnit

Press Enter to insert, Tab to replace  



小结

1. 常用的计量单位的使用

数据校验

- 开启数据校验有助于系统安全性，J2EE规范中JSR303规范定义了一组有关数据校验相关的API

步骤 开启Bean数据校验

①：添加JSR303规范坐标与Hibernate校验框架对应坐标

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
</dependency>

<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
</dependency>
```


步骤 开启Bean数据校验

②：对Bean开启校验功能

```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
@Validated
public class ServerConfig {
}
```

步骤 开启Bean数据校验

③：设置校验规则

```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
@Validated
public class ServerConfig {
    @Max(value = 400,message = "最大值不能超过400")
    private int port;
}
```



小结

1. 启用Bean属性校验

- 导入JSR303与Hibernate校验框架坐标
- 使用@Validated注解启用校验功能
- 使用具体校验规则规范数据校验格式

yaml语法规则

- 字面值表达方式

```
boolean: TRUE           #TRUE,true,True,FALSE,false,False均可
float: 3.14             #6.8523015e+5 #支持科学计数法
int: 123                #0b1010_0111_0100_1010_1110 #支持二进制、八进制、十六进制
null: ~                #使用~表示null
string: HelloWorld     #字符串可以直接书写
string2: "Hello World" #可以使用双引号包裹特殊字符
date: 2018-02-17       #日期必须使用yyyy-MM-dd格式
datetime: 2018-02-17T15:02:31+08:00 #时间和日期之间使用T连接,最后使用+代表时区
```



小结

1. 注意yaml文件中对于数字的定义支持进制书写格式，如需使用字符串请使用引号明确标注



总结

1. @ConfigurationProperties
2. 宽松绑定/松散绑定
3. 常用计量单位绑定
4. 数据校验



测试

- 加载测试专用属性
- 加载测试专用配置
- Web环境模拟测试
- 数据层测试回滚
- 测试用例数据设定

加载测试专用属性

- 在启动测试环境时可以通过properties参数设置测试环境专用的属性

```
@SpringBootTest(properties = {"test.prop=testValue1"})
public class PropertiesAndArgsTest {
    @Value("${test.prop}")
    private String msg;
    @Test
    void testProperties(){
        System.out.println(msg);
    }
}
```

- ◆ 优势：比多环境开发中的测试环境影响范围更小，仅对当前测试类有效

加载测试专用属性

- 在启动测试环境时可以通过args参数设置测试环境专用的传入参数

```
@SpringBootTest(args = {"--test.arg=testValue2"})
public class PropertiesAndArgsTest {
    @Value("${test.arg}")
    private String msg;
    @Test
    void testArgs(){
        System.out.println(msg);
    }
}
```



小结

1. 加载测试临时属性应用于小范围测试环境

加载测试专用配置

- 使用@Import注解加载当前测试类专用的配置

```
@SpringBootTest
@Import(MsgConfig.class)
public class ConfigurationTest {
    @Autowired
    private String msg;
    @Test
    void testConfiguration(){
        System.out.println(msg);
    }
}
```



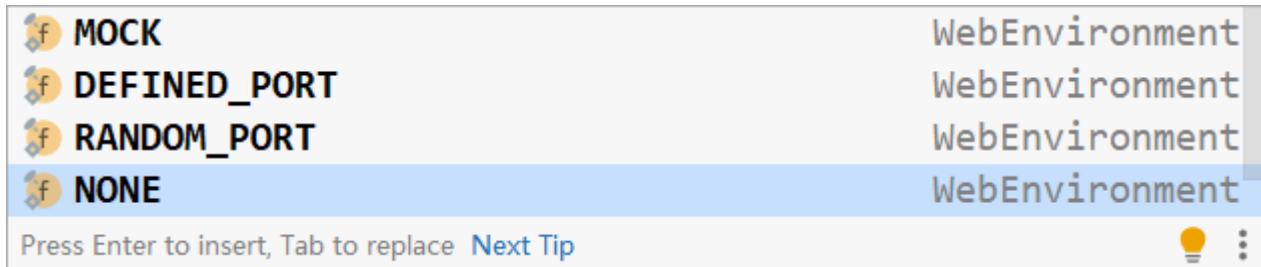
小结

1. 加载测试范围配置应用于小范围测试环境

web环境模拟测试

- 模拟端口

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class WebTest {
    @Test
    void testRandomPort () {
    }
}
```



web环境模拟测试

- 虚拟请求测试

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//开启虚拟MVC调用
@AutoConfigureMockMvc
public class WebTest {
    @Test
    //注入虚拟MVC调用对象
    public void testWeb(@Autowired MockMvc mvc) throws Exception {
        //创建虚拟请求, 当前访问/books
        MockHttpServletRequestBuilder builder = MockMvcRequestBuilders.get("/books");
        //执行请求
        ResultActions action = mvc.perform(builder);
    }
}
```

web环境模拟测试

- 虚拟请求状态匹配

```
@Test
public void testSataus(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder = MockMvcRequestBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //匹配执行状态 (是否预期值)
    //定义执行状态匹配器
    StatusResultMatchers status = MockMvcResultMatchers.status();
    //定义预期执行状态
    ResultMatcher ok = status.isOk();
    //使用本次真实执行结果与预期结果进行比对
    action.andExpect(ok);
}
```

web环境模拟测试

- 虚拟请求响应体匹配

```
@Test
public void testBody(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder = MockMvcRequestBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //匹配执行结果 (是否预期值)
    //定义执行结果匹配器
    ContentResultMatchers content = MockMvcResultMatchers.content();
    //定义预期执行结果
    ResultMatcher result = content().string("springboot");
    //使用本次真实执行结果与预期结果进行比对
    action.andExpect(result);
}
```


web环境模拟测试

- 虚拟请求响应体 (json) 匹配

```
@Test
public void testJson(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder = MockMvcRequestBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //匹配执行结果 (是否预期值)
    //定义执行结果匹配器
    ContentResultMatchers content = MockMvcResultMatchers.content();
    //定义预期执行结果
    ResultMatcher result = content.json("{\"id\":1,\"name\":\"SpringBoot2\"}");
    //使用本次真实执行结果与预期结果进行比对
    action.andExpect(result);
}
```

web环境模拟测试

- 虚拟请求响应头匹配

```
@Test
public void testContentType(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder = MockMvcRequestBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    HeaderResultMatchers header = MockMvcResultMatchers.header();
    ResultMatcher resultHeader = header.string("Content-Type", "application/json");
    action.andExpect(resultHeader);
}
```



小结

1. web环境模拟测试

- 设置测试端口
- 模拟测试启动
- 模拟测试匹配 (各组成部分信息均可匹配)

业务层测试事务回滚

- 为测试用例添加事务，SpringBoot会对测试用例对应的事务提交操作进行回滚

```
@SpringBootTest
@Transactional
public class DaoTest {
    @Autowired
    private BookService bookService;
}
```

- 如果想在测试用例中提交事务，可以通过@Rollback注解设置

```
@SpringBootTest
@Transactional
@Rollback(false)
public class DaoTest {
}
```



小结

1. 测试用例回滚事务

测试用例数据设定

- 测试用例数据通常采用随机值进行测试，使用SpringBoot提供的随机数为其赋值

```
testcast:
  book:
    id: ${random.int}           # 随机整数
    id2: ${random.int(10)}     # 10以内随机数
    type: ${random.int(10,20)} # 10到20随机数
    uuid: ${random.uuid}      # 随机uuid
    name: ${random.value}     # 随机字符串,MD5字符串,32位
    publishTime: ${random.long} # 随机整数(Long范围)
```

- ◆ `${random.int}`表示随机整数
- ◆ `${random.int(10)}`表示10以内的随机数
- ◆ `${random.int(10,20)}`表示10到20的随机数
- ◆ 其中()可以是任意字符，例如[]，!!均可



小结

1. 使用随机数据替换测试用例中书写固定的数据



总结

1. 加载测试专用属性
2. 加载测试专用配置
3. Web环境模拟测试
4. 数据层测试回滚
5. 测试用例数据设定



数据层解决方案

- SQL
- NoSQL

数据层解决方案

- 现有数据层解决方案技术选型

Druid + MyBatis-Plus + MySQL

- ◆ 数据源: DruidDataSource
- ◆ 持久化技术: MyBatis-Plus / MyBatis
- ◆ 数据库: MySQL

数据源配置格式

- 格式一

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
    username: root
    password: root
```

- 格式二

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root
```

数据源配置

- SpringBoot提供了3种内嵌数据库
 - ◆ HikariCP
 - ◆ Tomcat提供DataSource
 - ◆ Commons DBCP

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
    username: root
    password: root
```

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root
```

数据源配置

- SpringBoot提供了3种内嵌的数据源对象供开发者选择
 - ◆ HikariCP: 默认内置数据源对象
 - ◆ Tomcat提供DataSource: HikariCP不可用的情况下，且在web环境中，将使用tomcat服务器配置的数据源对象
 - ◆ Commons DBCP: Hikari不可用，tomcat数据源也不可用，将使用dbcp数据源
- 通用配置无法设置具体的数据源配置信息，仅提供基本的连接相关配置，如需配置，在下一级配置中设置具体设定

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
  hikari:
    maximum-pool-size: 50
```



小结

1. SpringBoot内置3款数据源可供选择

- HikariCP (默认)
- Tomcat提供DataSource
- Commons DBCP

数据层解决方案

- 现有数据层解决方案技术选型

Druid + MyBatis-Plus + MySQL

- ◆ 数据源: DruidDataSource
- ◆ 持久化技术: MyBatis-Plus / MyBatis
- ◆ 数据库: MySQL

数据层解决方案

- 内置持久化解决方案——JdbcTemplate

```
@SpringBootTest
class Springboot15SqlApplicationTests {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Test
    void testJdbc(){
        String sql = "select * from tbl_book where id = 1";
        List<Book> query = jdbcTemplate.query(sql, new RowMapper<Book>() {
            @Override
            public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
                Book temp = new Book();
                temp.setId(rs.getInt("id"));
                temp.setName(rs.getString("name"));
                temp.setType(rs.getString("type"));
                temp.setDescription(rs.getString("description"));
                return temp;
            }
        });
        System.out.println(query);
    }
}
```


数据层解决方案

- 内置持久化解决方案——JdbcTemplate

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

数据层解决方案

- JdbcTemplate配置

```
spring:
  jdbc:
    template:
      query-timeout: -1    # 查询超时时间
      max-rows: 500      # 最大行数
      fetch-size: -1     # 缓存行数
```



小结

1. SpringBoot内置JdbcTemplate持久化解决方案
2. 使用JdbcTemplate需要导入spring-boot-starter-jdbc

数据层解决方案

- 现有数据层解决方案技术选型

Druid + MyBatis-Plus + MySQL

- ◆ 数据源: DruidDataSource
- ◆ 持久化技术: MyBatis-Plus / MyBatis
- ◆ 数据库: MySQL

内嵌数据库

- SpringBoot提供了3种内嵌数据库供开发者选择，提高开发测试效率
 - ◆ H2
 - ◆ HSQL
 - ◆ Derby

内嵌数据库 (H2)

- 导入H2相关坐标

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

内嵌数据库 (H2)

- 设置当前项目为web工程，并配置H2管理控制台参数

```
server:  
  port: 80  
spring:  
  h2:  
    console:  
      path: /h2  
      enabled: true
```

- ◆ 访问用户名sa，默认密码123456

内嵌数据库 (H2)

- 操作数据库 (创建表)

```
create table tbl_book (id int,name varchar,type varchar,description varchar)
```


内嵌数据库 (H2)

- 设置访问数据源

```
server:  
  port: 80  
spring:  
  datasource:  
    driver-class-name: org.h2.Driver  
    url: jdbc:h2:~/test  
    username: sa  
    password: 123456  
  h2:  
    console:  
      path: /h2  
      enabled: true
```

内嵌数据库 (H2)

- H2数据库控制台仅用于开发阶段，线上项目请务必关闭控制台功能

```
server:  
  port: 80  
spring:  
  h2:  
    console:  
      path: /h2  
      enabled: false
```

数据库设定

- SpringBoot可以根据url地址自动识别数据库种类，在保障驱动类存在的情况下，可以省略配置

```
server:  
  port: 80  
spring:  
  datasource:  
#    driver-class-name: org.h2.Driver  
  url: jdbc:h2:~/test  
  username: sa  
  password: 123456  
h2:  
  console:  
    path: /h2  
    enabled: true
```



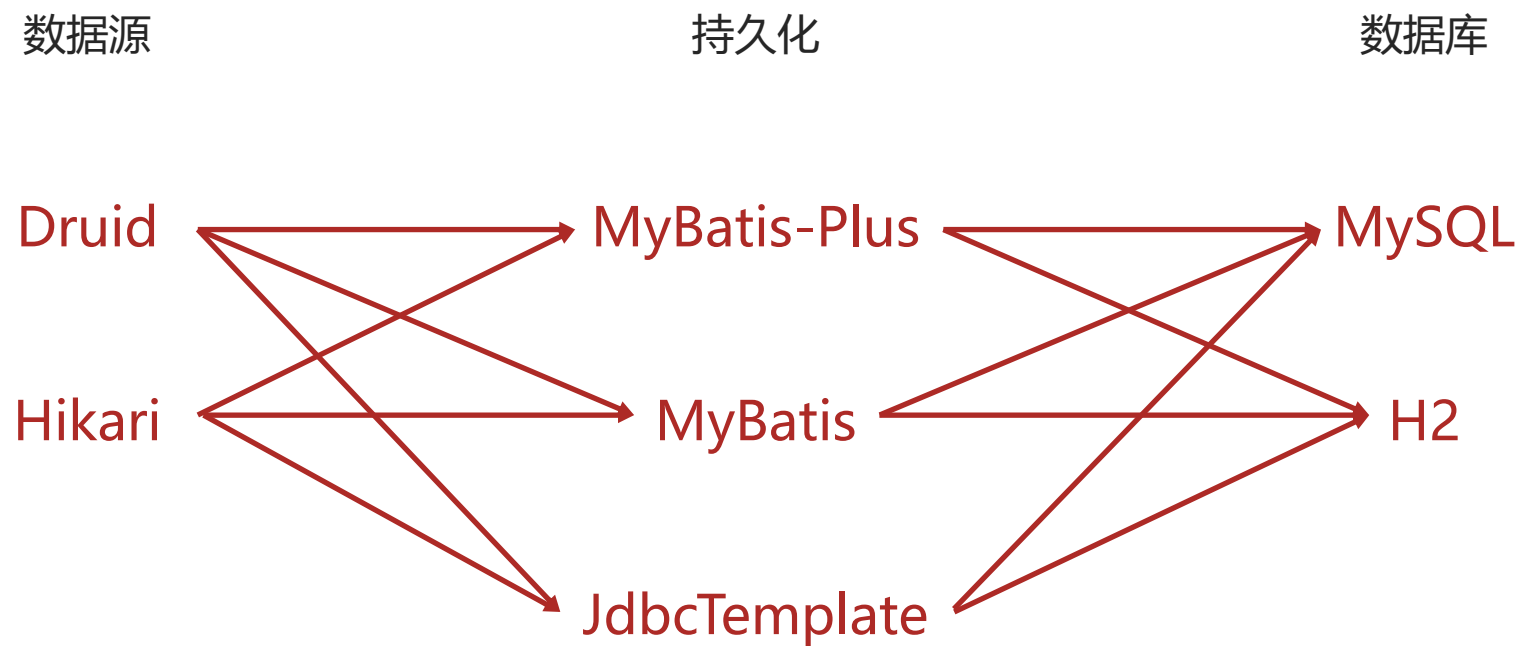
小结

1. H2内嵌式数据库启动方式
2. H2数据库线上运行时请务必关闭

数据层解决方案

- 现有数据层解决方案技术选型

Druid + MyBatis-Plus + MySQL





总结

1. 数据源配置 (Hikari)
2. 持久化技术 (JdbcTemplate)
3. 数据库 (H2)



数据层解决方案

- SQL
- NoSQL

NoSQL解决方案

- 市面上常见的NoSQL解决方案
 - ◆ Redis
 - ◆ Mongo
 - ◆ ES
 - ◆ Solr
- 说明：上述技术通常在Linux系统中安装部署，为降低学习者压力，本课程制作基于Windows版安装所有的软件并基于Windows版安装的软件进行课程制作

Redis

- Redis是一款key-value存储结构的内存级NoSQL数据库
 - ◆ 支持多种数据存储格式
 - ◆ 支持持久化
 - ◆ 支持集群
- Redis下载（Windows版）
 - ◆ <https://github.com/tporadowski/redis/releases>
- Redis安装与启动（Windows版）
 - ◆ Windows解压安装或一键式安装
 - ◆ 服务端启动命令

```
redis-server.exe redis.windows.conf
```

- ◆ 客户端启动命令

```
redis-cli.exe
```



小结

1. Redis安装 (Windows版)
2. Redis基本操作

Redis

- 导入SpringBoot整合Redis坐标

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

Redis

- 配置Redis (采用默认配置)

```
spring:  
  redis:  
    host: localhost # 127.0.0.1  
    port: 6379
```

- ◆ 主机: localhost (默认)
- ◆ 端口: 6379 (默认)

Redis

- RedisTemplate提供操作各种数据存储类型的接口API

m opsForCluster()	ClusterOperations
m opsForValue()	ValueOperations
m opsForGeo()	GeoOperations
m opsForHash()	HashOperations
m opsForList()	ListOperations
m opsForHyperLogLog()	HyperLogLogOperations
m opsForSet()	SetOperations
m opsForStream()	StreamOperations
m opsForStream(HashMapmer hashMappe...	StreamOperations
m opsForZSet()	ZSetOperations

Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor [Next Tip](#) ⋮

Redis

- 客户端: RedisTemplate

```
@SpringBootTest
class Springboot16NosqlApplicationTests {
    @Test
    void set(@Autowired RedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set("testKey", "testValue");
    }
    @Test
    void get(@Autowired RedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        Object val = ops.get("testKey");
        System.out.println(val);
    }
}
```

Redis

- 客户端: RedisTemplate

```
@SpringBootTest
class Springboot16NosqlApplicationTests {
    @Test
    void set(@Autowired RedisTemplate redisTemplate) {
        HashOperations opsH = redisTemplate.opsForHash();
        opsH.put("testKeyH", "testFieldH", "testValueH");
    }
    @Test
    void get(@Autowired RedisTemplate redisTemplate) {
        HashOperations opsH = redisTemplate.opsForHash();
        Object valH = opsH.get("testKeyH", "testFieldH");
        System.out.println(valH);
    }
}
```



小结

1. SpringBoot整合Redis

- 导入redis对应的starter
- 配置
- 提供操作Redis接口对象RedisTemplate
 - ◆ ops*: 获取各种数据类型操作接口

Redis

- 客户端：RedisTemplate以对象作为key和value，内部对数据进行序列化

```
@SpringBootTest
class Springboot16NosqlApplicationTests {
    @Test
    void set(@Autowired RedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set("testKey", "testValue");
    }
    @Test
    void get(@Autowired RedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        Object val = ops.get("testKey");
        System.out.println(val);
    }
}
```

Redis

- 客户端：StringRedisTemplate以字符串作为key和value，与Redis客户端操作等效

```
@SpringBootTest
class Springboot16NosqlApplicationTests {
    @Test
    void set(@Autowired StringRedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set("testKey", "testValue");
    }
    @Test
    void get(@Autowired StringRedisTemplate redisTemplate) {
        ValueOperations ops = redisTemplate.opsForValue();
        Object val = ops.get("testKey");
        System.out.println(val);
    }
}
```



小结

1. RedisTemplate
2. StringRedisTemplate (常用)

Redis

- 客户端选择: jedis

```
<dependency>  
  <groupId>redis.clients</groupId>  
  <artifactId>jedis</artifactId>  
</dependency>
```

Redis

- 配置客户端

```
spring:  
  redis:  
    host: localhost # 127.0.0.1  
    port: 6379  
    client-type: Lettuce
```

Redis

- 配置客户端专用属性

```
spring:
  redis:
    host: localhost # 127.0.0.1
    port: 6379
    client-type: Lettuce
    lettuce:
      pool:
        max-active: 16
    jedis:
      pool:
        max-active: 16
```

Redis

- lettucus与jedis区别
 - jedis连接Redis服务器是直连模式，当多线程模式下使用jedis会存在线程安全问题，解决方案可以通过配置连接池使每个连接专用，这样整体性能就大受影响。
 - lettucus基于Netty框架进行与Redis服务器连接，底层设计中采用StatefulRedisConnection。StatefulRedisConnection自身是线程安全的，可以保障并发访问安全问题，所以一个连接可以被多线程复用。当然lettucus也支持多连接实例一起工作。



小结

1. SpringBoot整合Redis客户端选择

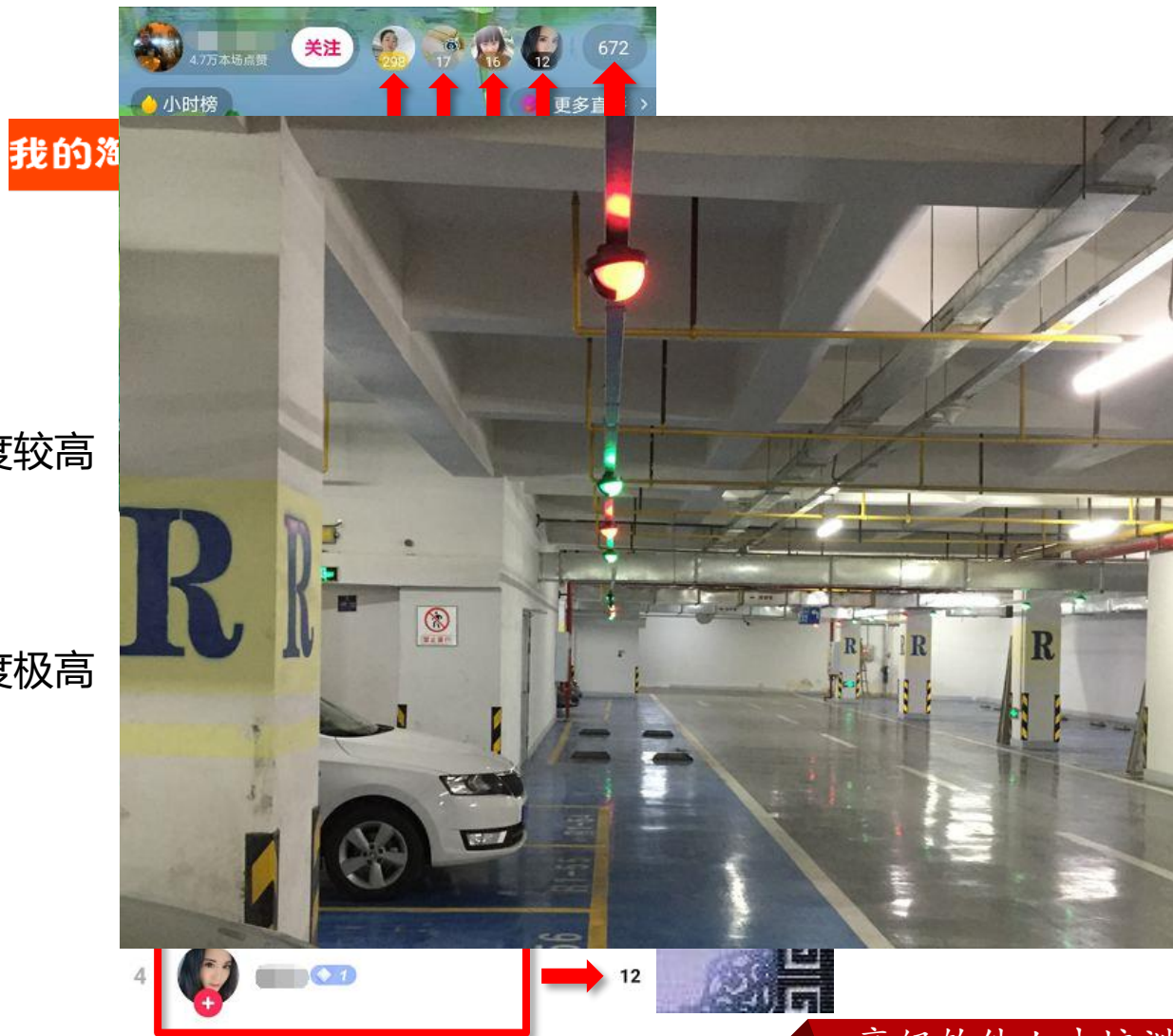
- lettuce (默认)
- jedis

Mongodb

- MongoDB是一个开源、高性能、无模式的文档型数据库。NoSQL数据库产品中的一种，是最像关系型数据库的非关系型数据库

Mongodb

- 淘宝用户数据
 - ◆ 存储位置：数据库
 - ◆ 特征：永久性存储，修改频度极低
- 游戏装备数据、游戏道具数据
 - ◆ 存储位置：数据库、Mongodb
 - ◆ 特征：永久性存储与临时存储相结合、修改频度较高
- 直播数据、打赏数据、粉丝数据
 - ◆ 存储位置：数据库、Mongodb
 - ◆ 特征：永久性存储与临时存储相结合，修改频度极高
- 物联网数据
 - ◆ 存储位置：Mongodb
 - ◆ 特征：临时存储，修改频度飞速
- 其他数据.....





小结

1. Mongodb应用场景

Mongodb

- Windows版Mongo下载
 - ◆ <https://www.mongodb.com/try/download>
- Windows版Mongo安装
 - ◆ 解压缩后设置数据目录
- Windows版Mongo启动
 - ◆ 服务端启动

```
mongod --dbpath=..\data\db
```

- ◆ 客户端启动

```
mongo --host=127.0.0.1 --port=27017
```

Mongodb

- Windows版Mongo安装问题及解决方案

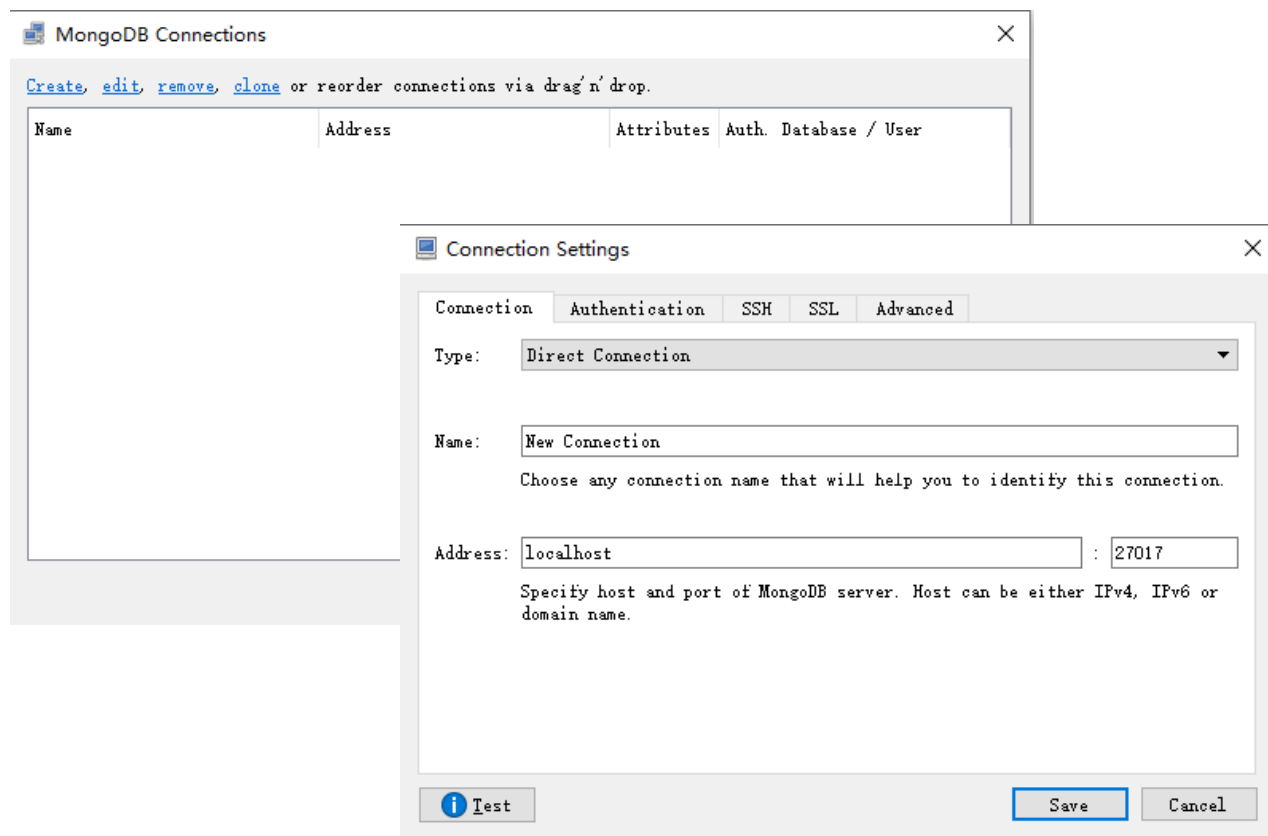


- ◆ 步骤一：下载对应的dll文件（通过互联网搜索即可）
- ◆ 步骤二：拷贝到windows安装路径下的system32目录中
- ◆ 步骤三：执行命令注册对应dll文件

```
regsvr32 vcruntime140_1.dll
```

Mongodb

- 可视化客户端——Robo 3T





小结

1. Mongodb安装、启动
2. 可视化客户端Robo 3T安装与连接

Mongodb

- 新增

```
db.集合名称.insert/save/insertOne(文档)
```

- 修改

```
db.集合名称.remove(条件)
```

- 删除

```
db.集合名称.update(条件, {操作种类:{文档}})
```


Mongodb

1. 基础查询

- ◆ 查询全部: `db.集合.find()`;
- ◆ 查第一条: `db.集合.findOne()`
- ◆ 查询指定数量文档: `db.集合.find().limit(10)` //查10条文档
- ◆ 跳过指定数量文档: `db.集合.find().skip(20)` //跳过20条文档
- ◆ 统计: `db.集合.count()`
- ◆ 排序: `db.集合.sort({age:1})` //按age升序排序
- ◆ 投影: `db.集合名称.find(条件,{name:1,age:1})` //仅保留name与age域

2. 条件查询

- ◆ 基本格式: `db.集合.find({条件})`
- ◆ 模糊查询: `db.集合.find({域名:/正则表达式/})` //等同SQL中的like, 比like强大, 可以执行正则所有规则
- ◆ 条件比较运算: `db.集合.find({域名:{$gt:值}})` //等同SQL中的数值比较操作, 例如: `name>18`
- ◆ 包含查询: `db.集合.find({域名:{$in:[值1, 值2]}})` //等同于SQL中的in
- ◆ 条件连接查询: `db.集合.find({$and:[{条件1},{条件2]}})` //等同于SQL中的and、or



小结

1. Mongodb基础CRUD

Mongodb

- 导入Mongodb驱动

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

Mongodb

- 配置客户端

```
spring:  
  data:  
    mongodb:  
      uri: mongodb://localhost/itheima
```

Mongodb

- 客户端读写Mongodb

```
@Test
void testSave(@Autowired MongoTemplate mongoTemplate){
    Book book = new Book();
    book.setId(1);
    book.setType("springboot");
    book.setName("springboot");
    book.setDescription("springboot");
    mongoTemplate.save(book);
}

@Test
void testFind(@Autowired MongoTemplate mongoTemplate){
    List<Book> all = mongoTemplate.findAll(Book.class);
    System.out.println(all);
}
```



小结

1. SpringBoot整合Mongodb

- 导入Mongodb对应的starter
- 配置mongodb访问uri
- 提供操作Mongodb接口对象MongoTemplate

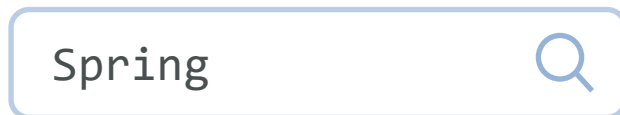
ElasticSearch (ES)

- Elasticsearch是一个分布式全文搜索引擎



ElasticSearch (ES)

- Elasticsearch是一个分布式全文搜索引擎



id	type	name	description
1	计算机理论	Spring 实战 第5版	Spring 入门经典教程，深入理解 Spring 原理技术内幕
2	计算机理论	Spring 5核心原理与30个类手写实战	十年沉淀之作，手写 Spring 精华思想
3	计算机理论	Spring 5 设计模式	深入 Spring 源码剖析 Spring 源码中蕴含的10大设计模式
4	计算机理论	Spring MVC+MyBatis开发从入门到项目实战	全方位解析面向Web应用的轻量级框架，带你成为 Spring MVC开
5	计算机理论	轻量级Java Web企业应用实战	源码级剖析 Spring 框架，适合已掌握Java基础的读者
6	计算机理论	Java核心技术 卷I 基础知识 (原书第11版)	Core Java 第11版，Jolt大奖获奖作品，针对Java SE9、10、11等
7	计算机理论	深入理解Java虚拟机	5个维度全面剖析JVM，大厂面试知识点全覆盖
8	计算机理论	Java编程思想 (第4版)	Java学习必读经典殿堂级著作！赢得了全球程序员的广泛赞誉
9	计算机理论	零基础学Java (全彩版)	零基础自学编程的入门图书，由浅入深，详解Java语言的编程思想
10	市场营销	直播就该这么做：主播高效沟通实战指南	李子柒、李佳琦、薇娅成长为网红的秘密都在书中
11	市场营销	直播销讲实战一本通	和秋叶一起学系列网络营销书籍
12	市场营销	直播带货：淘宝、天猫直播从新手到高手	一本教你如何玩转直播的书，10堂课轻松实现带货月入3W+

ElasticSearch (ES)

- Elasticsearch是一个分布式全文搜索引擎

Spring

1 计算机理论 Spring实战 第5版

Spring 实战 第5版



TOMSPower托马斯电动割...
店铺被8千人种草 | 修剪机除草
¥531.00
领券每满200减30
一年评价200+ 98%好评
TOMSPower电动工具官... 进店>



德国骏威 (Junwei) 家用多功能
288VF手电钻五金工具箱套装...
手电钻 切割机 冲击扳手
¥208.00
京东物流 京东超市 京贴200减30
1000+条评价 100%好评
骏威欣锐专卖店 进店>



启一无刷手电钻充电电钻家用电
动螺丝刀自电营转钻墙充电钻...
礼包 套装 充电
¥99.00
5000+条评价 98%好评
启一官方旗舰店 进店>



【专业款】准心手电钻锂电电钻
电动螺丝刀家用充电式手钻电转...
充电式 打孔 套装
¥168.00
京东物流 京东超市 领券130减20
1万+条评价 97%好评
京东好店 准心精度专卖店 进店



全铜电机 轻巧强劲 包用10年
预估价 ¥198 = 京东价 ¥208 - 促销 ¥10

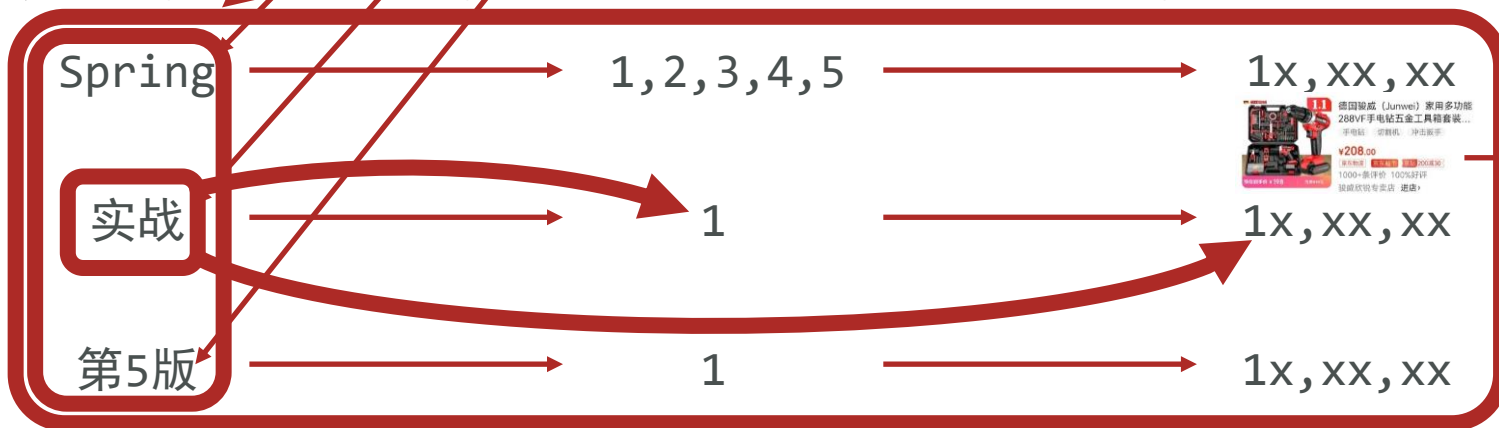
¥208.00
领立减80元优惠券, 及12期免息券 >
京东超市 一站式囤生活好物 >
京东物流 德国骏威 (Junwei) 家用多功能
288VF手电钻五金工具箱套装家庭电工木工电
讯维修工具修理组合套装 尊享荣耀王者升级
版锂电套装
7天价保 电动园艺刀具热卖榜第8名

ElasticSearch (ES)

- Elasticsearch是一个分布式全文搜索引擎

索引 倒排索引 创建文档 使用文档

id	type	name	description
1	计算机理论	Spring实战 第5版	Spring入门经典教程，深入理解Spring原理技术内幕
2	计算机理论	Spring 5核心原理与30个类手写实战	十年沉淀之作，手写Spring精华思想
3	计算机理论	Spring 5 设计模式	深入Spring源码剖析Spring源码中蕴含的10大设计模式
4	计算机理论	Spring MVC+MyBatis开发从入门到项目实战	全方位解析面向Web应用的轻量级框架，带你成为Spring MVC专家
5	计算机理论	轻量级Java Web企业应用实战	源码级剖析Spring框架，适合已掌握Java基础的读者
6	计算机理论	Java核心技术 卷I 基础知识 (原书第11版)	Core Java 第11版，Jolt大奖获奖作品，针对Java SE9、10、11





小结

1. ES应用场景
2. ES相关概念

ElasticSearch (ES)

- Windows版ES下载
 - ◆ <https://www.elastic.co/cn/downloads/elasticsearch>
- Windows版ES安装与启动

```
运行 elasticsearch.bat
```



小结

1. ES下载与安装

ElasticSearch (ES)

- 创建/查询/删除索引

```
PUT      http://localhost:9200/books
```

```
GET      http://localhost:9200/books
```

```
DELETE   http://localhost:9200/books
```

ElasticSearch (ES)

- IK分词器
 - ◆ 下载: <https://github.com/medcl/elasticsearch-analysis-ik/releases>

ElasticSearch (ES)

- 创建索引并指定规则

```
{
  "mappings":{
    "properties":{
      "id":{
        "type":"keyword"
      },
      "name":{
        "type":"text",      "analyzer":"ik_max_word",      "copy_to":"all"
      },
      "type":{
        "type":"keyword"
      },
      "description":{
        "type":"text",      "analyzer":"ik_max_word",      "copy_to":"all"
      },
      "all":{
        "type":"text",      "analyzer":"ik_max_word"
      }
    }
  }
}
```




小结

1. 索引操作
2. IK分词器安装
3. 设置索引创建规则 (应用)

ElasticSearch (ES)

- 创建文档

POST	http://localhost:9200/books/_doc	#使用系统生成id
POST	http://localhost:9200/books/_create/1	#使用指定id,
POST	http://localhost:9200/books/_doc/1	#使用指定id, 不存在创建, 存在更新 (版本递增)

```
{  
  "name":"springboot",  
  "type":"springboot",  
  "description":"springboot"  
}
```

ElasticSearch (ES)

- 查询文档

```
GET      http://localhost:9200/books/_doc/1      #查询单个文档
GET      http://localhost:9200/books/_search  #查询全部文档
```

- 条件查询

```
GET      http://localhost:9200/books/_search?q=name:springboot
```

- 删除文档

```
DELETE   http://localhost:9200/books/_doc/1
```

ElasticSearch (ES)

- 修改文档 (全量修改)

```
PUT      http://localhost:9200/books/_doc/1
```

```
{  
  "name":"springboot",  
  "type":"springboot",  
  "description":"springboot"  
}
```

- 修改文档 (部分修改)

```
POST     http://localhost:9200/books/_update/1
```

```
{  
  "doc":{  
    "name":"springboot"  
  }  
}
```



小结

1. 文档操作

- 增删改查

ElasticSearch (ES)

- 导入坐标

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

- 配置

```
spring:
  elasticsearch:
    rest:
      uris: http://localhost:9200
```

- 客户端

```
@SpringBootTest
class Springboot18EsApplicationTests {
    @Autowired
    private ElasticsearchRestTemplate template;
}
```

ElasticSearch (ES)

- SpringBoot平台并没有跟随ES的更新速度进行同步更新，ES提供了High Level Client操作ES
- 导入坐标

```
<dependency>  
  <groupId>org.elasticsearch.client</groupId>  
  <artifactId>elasticsearch-rest-high-level-client</artifactId>  
</dependency>
```

- 配置 (无)

ElasticSearch (ES)

- 客户端

```
@Test
void test() throws IOException {
    HttpHost host = HttpHost.create("http://localhost:9200");
    RestClientBuilder builder = RestClient.builder(host);
    RestHighLevelClient client = new RestHighLevelClient(builder);
    //客户端操作
    CreateIndexRequest request = new CreateIndexRequest("books");
    //获取操作索引的客户端对象，调用创建索引操作
    client.indices().create(request, RequestOptions.DEFAULT);
    //关闭客户端
    client.close();
}
```


ElasticSearch (ES)

- 客户端改进

```
@SpringBootTest
class Springboot18EsApplicationTests {
    @Autowired
    private BookDao bookDao;
    @Autowired
    RestHighLevelClient client;
    @BeforeEach
    void setUp() {
        this.client = new RestHighLevelClient(RestClient.builder(HttpHost.create("http://localhost:9200")));
    }

    @AfterEach
    void tearDown() throws IOException {
        this.client.close();
    }
}
```

ElasticSearch (ES)

- 客户端改进

```
@Test
void test() throws IOException {
    //客户端操作
    CreateIndexRequest request = new CreateIndexRequest("books");
    //获取操作索引的客户端对象，调用创建索引操作
    client.indices().create(request, RequestOptions.DEFAULT);
}
```



小结

1. High Leven Client
2. 客户端初始化
3. 客户端改进

ElasticSearch (ES)

- 创建索引

```
// 创建索引
@Test
void testCreateIndexByIK() throws IOException {
    HttpHost host = HttpHost.create("http://localhost:9200");
    RestClientBuilder builder = RestClient.builder(host);
    RestHighLevelClient client = new RestHighLevelClient(builder);
    // 客户端操作
    CreateIndexRequest request = new CreateIndexRequest("b");
    // 设置要执行操作
    String json = "";
    // 设置请求参数，参数类型json数据
    request.source(json, XContentType.JSON);
    // 获取操作索引的客户端对象，调用创建索引操作
    client.indices().create(request, RequestOptions.DEFAULT);
    // 关闭客户端
    client.close();
}
```

```
String json = "{\n" +
    "  \"mappings\":{\n" +
    "    \"properties\":{\n" +
    "      \"id\":{\n" +
    "        \"type\": \"keyword\"\n" +
    "      },\n" +
    "      \"name\":{\n" +
    "        \"type\": \"text\", \n" +
    "        \"analyzer\": \"ik_max_word\", \n" +
    "        \"copy_to\": \"all\"\n" +
    "      },\n" +
    "      \"type\":{\n" +
    "        \"type\": \"keyword\"\n" +
    "      },\n" +
    "      \"description\":{\n" +
    "        \"type\": \"text\", \n" +
    "        \"analyzer\": \"ik_max_word\", \n" +
    "        \"copy_to\": \"all\"\n" +
    "      },\n" +
    "      \"all\":{\n" +
    "        \"type\": \"text\", \n" +
    "        \"analyzer\": \"ik_max_word\"\n" +
    "      }\n" +
    "    }\n" +
    "  }\n" +
    "};"
```

ElasticSearch (ES)

- 添加文档

```
//添加文档
@Test
void testCreateDoc() throws IOException {
    Book book = bookDao.selectById(1);
    IndexRequest request = new IndexRequest("books").id(book.getId().toString());
    String json = JSON.toJSONString(book);
    request.source(json, XContentType.JSON);
    client.index(request, RequestOptions.DEFAULT);
}
```

ElasticSearch (ES)

- 批量添加文档

```
// 批量添加文档
@Test
void testCreateDocAll() throws IOException {
    List<Book> bookList = bookDao.selectList(null);
    BulkRequest bulk = new BulkRequest();
    for (Book book : bookList) {
        IndexRequest request = new IndexRequest("books").id(book.getId().toString());
        String json = JSON.toJSONString(book);
        request.source(json, XContentType.JSON);
        bulk.add(request);
    }
    client.bulk(bulk, RequestOptions.DEFAULT);
}
```



小结

1. 创建索引
2. 添加文档
3. 批量添加文档

ElasticSearch (ES)

- 按id查询文档

```
@Test
void testGet() throws IOException {
    GetRequest request = new GetRequest("books","1");
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    String json = response.getSourceAsString();
    System.out.println(json);
}
```


ElasticSearch (ES)

- 按条件查询文档

```
@Test
void testSearch() throws IOException {
    SearchRequest request = new SearchRequest("books");
    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.query(QueryBuilders.termQuery("all", "java"));
    request.source(builder);
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);
    SearchHits hits = response.getHits();
    for (SearchHit hit : hits) {
        String source = hit.getSourceAsString();
        Book book = JSON.parseObject(source, Book.class);
        System.out.println(book);
    }
}
```



小结

1. 根据id查询文档
2. 根据条件查询文档



总结

1. Redis
2. MongoDB
3. Elasticsearch

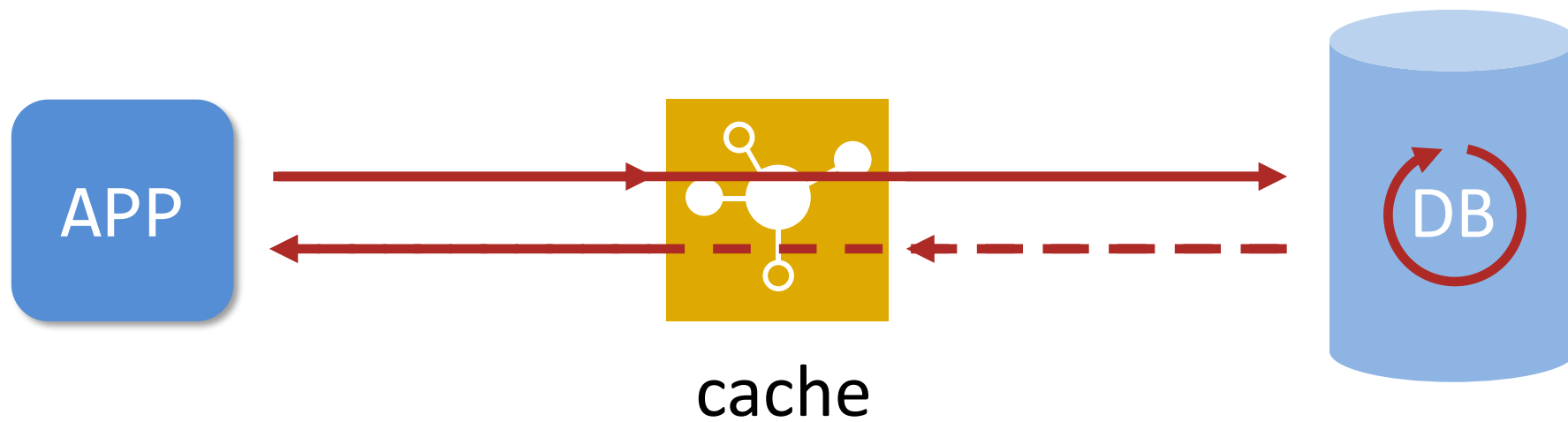


整合第三方技术

- 缓存
- 任务
- 邮件
- 消息

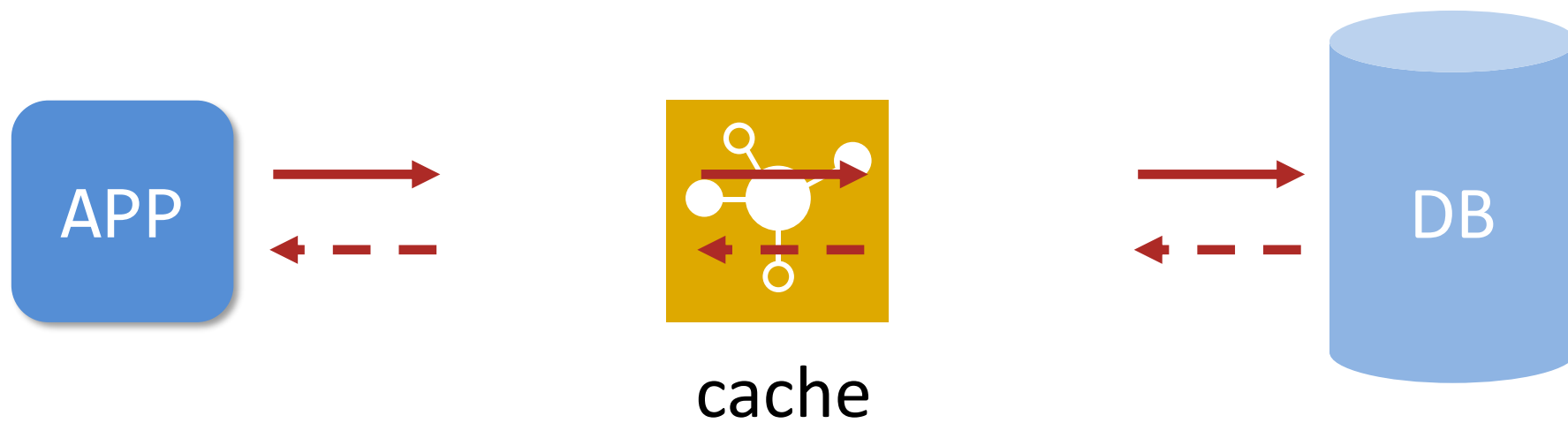
缓存

- 缓存是一种介于数据永久存储介质与数据应用之间的数据临时存储介质



缓存

- 缓存是一种介于数据永久存储介质与数据应用之间的数据临时存储介质
- 使用缓存可以有效的减少低速数据读取过程的次数（例如磁盘IO），提高系统性能
- 缓存不仅可以用于提高永久性存储介质的数据读取效率，还可以提供临时的数据存储空间





小结

1. 缓存作用
2. 自定义缓存

缓存

- SpringBoot提供了缓存技术，方便缓存使用

缓存使用

- 启用缓存
- 设置进入缓存的数据
- 设置读取缓存的数据

缓存使用

- 导入缓存技术对应的starter

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

缓存使用

- 启用缓存

```
@SpringBootApplication
@EnableCaching
public class Springboot19CacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot19CacheApplication.class, args);
    }
}
```

缓存使用

- 设置当前操作的结果数据进入缓存

```
@Cacheable(value="cacheSpace",key="#id")  
public Book getById(Integer id) {  
    return bookDao.selectById(id);  
}
```



小结

1. SpringBoot启用缓存的方式

- @EnableCaching
- @Cacheable

缓存

- SpringBoot提供的缓存技术除了提供默认的缓存方案，还可以对其他缓存技术进行整合，统一接口，方便缓存技术的开发与管理

缓存

- SpringBoot提供的缓存技术除了提供默认的缓存方案，还可以对其他缓存技术进行整合，统一接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)

缓存

- SpringBoot提供的缓存技术除了提供默认的缓存方案，还可以对其他缓存技术进行整合，统一接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存

- SpringBoot提供的缓存技术除了提供默认的缓存方案，还可以对其他缓存技术进行整合，统一接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存使用案例——手机验证码

- 需求
 - ◆ 输入手机号获取验证码，组织文档以短信形式发送给用户（页面模拟）
 - ◆ 输入手机号和验证码验证结果
- 需求分析
 - ◆ 提供controller，传入手机号，业务层通过手机号计算出独有的6位验证码数据，存入缓存后返回此数据
 - ◆ 提供controller，传入手机号与验证码，业务层通过手机号从缓存中读取验证码与输入验证码进行比对，返回比对结果

缓存使用案例——手机验证码

- 开启缓存

```
@SpringBootApplication
@EnableCaching
public class Springboot19CacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot19CacheApplication.class, args);
    }
}
```

缓存使用案例——手机验证码

- 业务层接口

```
public interface SMSCodeService {  
    /**  
     * 传入手机号获取验证码，存入缓存  
     * @param tele  
     * @return  
     */  
    String sendCodeToSMS(String tele);  
  
    /**  
     * 传入手机号与验证码，校验匹配是否成功  
     * @param smsCode  
     * @return  
     */  
    boolean checkCode(SMSCode smsCode);  
}
```

缓存使用案例——手机验证码

- 业务层设置获取验证码操作，并存储缓存，手机号为key，验证码为value

```
@Autowired
private CodeUtils codeUtils;
@CachePut(value = "smsCode",key="#tele")
public String sendCodeToSMS(String tele) {
    String code = codeUtils.generator(tele);
    return code;
}
```

缓存使用案例——手机验证码

- 业务层设置校验验证码操作，校验码通过缓存读取，返回校验结果

```
@Autowired
private CodeUtils codeUtils;
public boolean checkCode(SMSCode smsCode) {
    //取出内存中的验证码与传递过来的验证码比对，如果相同，返回true
    String code = smsCode.getCode();
    String cacheCode = codeUtils.get(smsCode.getTele());
    return code.equals(cacheCode);
}
```

```
@Component
public class CodeUtils {
    @Cacheable(value = "smsCode",key="#tele")
    public String get(String tele){
        return null;
    }
}
```



小结

1. 手机验证码案例实现

缓存

- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存供应商变更: Ehcache

- 加入Ehcache坐标 (缓存供应商实现)

```
<dependency>  
  <groupId>net.sf.ehcache</groupId>  
  <artifactId>ehcache</artifactId>  
</dependency>
```

缓存供应商变更：Ehcache

- 缓存设定为使用Ehcache

```
spring:  
  cache:  
    type: ehcache  
  ehcache:  
    config: ehcache.xml
```

缓存供应商变更: Ehcache

- 提供ehcache配置文件ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        updateCheck="false">
  <!-- 默认缓存策略 -->
  <!-- external: 是否永久存在, 设置为true则不会被清除, 此时与timeout冲突, 通常设置为false-->
  <!-- diskPersistent: 是否启用磁盘持久化-->
  <!-- maxElementsInMemory: 最大缓存数量-->
  <!-- overflowToDisk: 超过最大缓存数量是否持久化到磁盘-->
  <!-- timeToIdleSeconds: 最大不活动间隔, 设置过长缓存容易溢出, 设置过短无效果-->
  <!-- timeToLiveSeconds: 最大存活时间-->
  <!-- memoryStoreEvictionPolicy: 缓存清除策略-->
  <defaultCache
    eternal="false"
    diskPersistent="false"
    maxElementsInMemory="1000"
    overflowToDisk="false"
    timeToIdleSeconds="60"
    timeToLiveSeconds="60"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>
```

缓存供应商变更: Ehcache

- 提供ehcache配置文件ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        updateCheck="false">
  <cache
    name="smsCode"
    eternal="false"
    diskPersistent="false"
    maxElementsInMemory="1000"
    overflowToDisk="false"
    timeToIdleSeconds="10"
    timeToLiveSeconds="10"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>
```



小结

1. 变更缓存供应商为Ehcache

缓存

- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存供应商变更：Redis

- 加入Redis坐标（缓存供应商实现）

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

缓存供应商变更：Redis

- 配置Redis服务器，缓存设定为使用Redis

```
spring:  
  redis:  
    host: localhost  
    port: 6379  
  cache:  
    type: redis
```


缓存供应商变更：Redis

- 设置Redis相关配置

```
spring:
  redis:
    host: localhost
    port: 6379
  cache:
    type: redis
    redis:
      use-key-prefix: true      # 是否使用前缀名（系统定义前缀名）
      key-prefix: sms_         # 追加自定义前缀名
      time-to-live: 10s       # 有效时长
      cache-null-values: false # 是否允许存储空值
```



小结

1. 变更缓存供应商为Redis

缓存

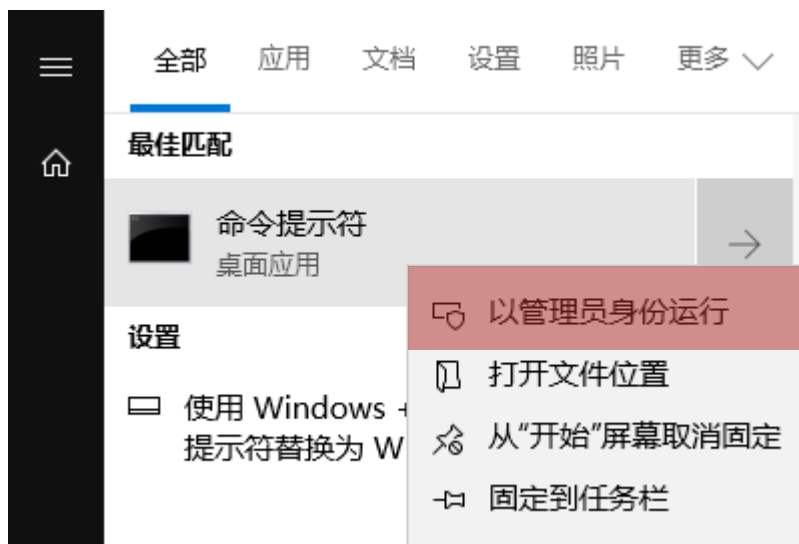
- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存供应商变更:memcached

- 下载memcached
 - ◆ 地址: <https://www.runoob.com/memcached/window-install-memcached.html>

缓存供应商变更:memcached

- 安装memcached
 - ◆ 使用管理员身份运行cmd指令



- ◆ 安装

```
memcached.exe -d install
```

缓存供应商变更:memcached

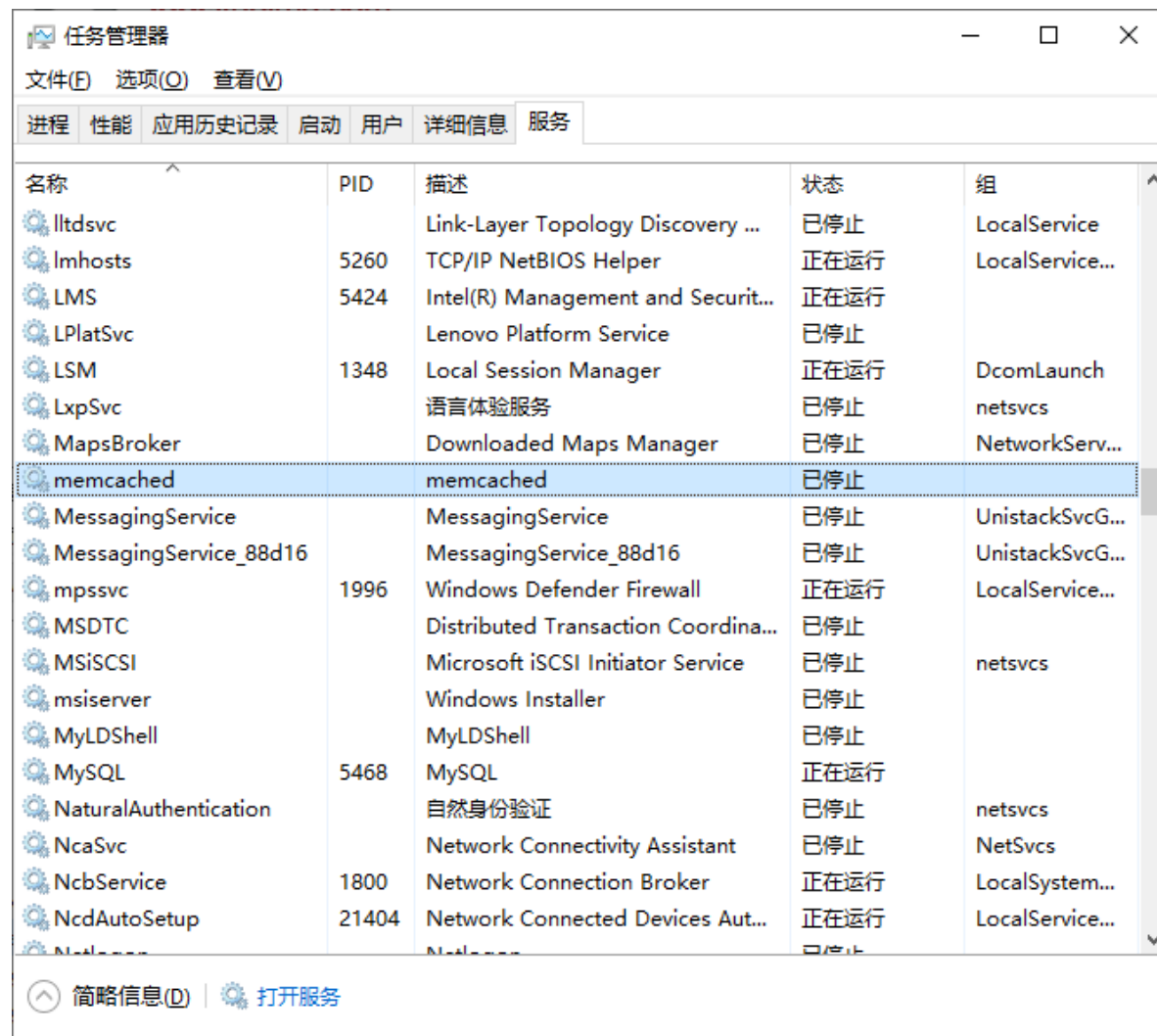
- 运行memcached

 - ◆ 启动服务

```
memcached.exe -d start
```

 - ◆ 停止服务

```
memcached.exe -d stop
```





小结

1. memcached下载与安装
2. memcached启动与退出

缓存供应商变更:memcached

- memcached客户端选择
 - ◆ Memcached Client for Java: 最早期客户端, 稳定可靠, 用户群广
 - ◆ SpyMemcached: 效率更高
 - ◆ Xmemcached: 并发处理更好
- SpringBoot未提供对memcached的整合, 需要使用硬编码方式实现客户端初始化

缓存供应商变更:memcached

- 加入Xmemcache坐标 (缓存供应商实现)

```
<dependency>  
  <groupId>com.googlecode.xmemcached</groupId>  
  <artifactId>xmemcached</artifactId>  
  <version>2.4.7</version>  
</dependency>
```

缓存供应商变更:memcached

- 配置memcached服务器必要属性

```
memcached:  
    # memcached服务器地址  
servers: localhost:11211  
    # 连接池的数量  
poolSize: 10  
    # 设置默认操作超时  
opTimeout: 3000
```

缓存供应商变更:memcached

- 创建读取属性配置信息类，加载配置

```
@Component
@ConfigurationProperties(prefix = "memcached")
@Data
public class XMemcachedProperties {
    private String servers;
    private Integer poolSize;
    private Long opTimeout;
}
```

缓存供应商变更:memcached

- 创建客户端配置类

```
@Configuration
public class XMemcachedConfig {
    @Autowired
    private XMemcachedProperties xMemcachedProperties;
    @Bean
    public MemcachedClient getMemcachedClient() throws IOException {
        MemcachedClientBuilder builder = new XMemcachedClientBuilder(xMemcachedProperties.getServers());
        MemcachedClient memcachedClient = builder.build();
        return memcachedClient;
    }
}
```

缓存供应商变更:memcached

- 配置memcached属性

```
@Service
public class SMSCodeServiceMemcacheImpl implements SMSCodeService {
    @Autowired
    private CodeUtils codeUtils;
    @Autowired
    private MemcachedClient memcachedClient;
    @Override
    public String sendCodeToSMS(String tele) {
        String code = this.codeUtils.generator(tele);
        //将数据加入memcache
        try {
            memcachedClient.set(tele,0,code);           // key,timeout,value
        } catch (Exception e) {
            e.printStackTrace();
        }
        return code;
    }
}
```

缓存供应商变更:memcached

- 配置memcached属性

```
@Service
public class SMSCodeServiceMemcacheImpl implements SMSCodeService {
    @Autowired
    private CodeUtils codeUtils;
    @Autowired
    private MemcachedClient memcachedClient;
    @Override
    public boolean checkCode(CodeMsg codeMsg) {
        String value = null;
        try {
            value = memcachedClient.get(codeMsg.getTele()).toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return codeMsg.getCode().equals(value);
    }
}
```



小结

1. xmemcached客户端加载方式 (bean初始化)
2. xmemcached客户端使用方式 (set & get)

缓存

- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached

缓存

- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached
 - ◆ jcache (阿里)

缓存供应商变更:jetcache

- jetCache对SpringCache进行了封装，在原有功能基础上实现了多级缓存、缓存统计、自动刷新、异步调用、数据报表等功能
- jetCache设定了本地缓存与远程缓存的多级缓存解决方案
 - ◆ 本地缓存 (local)
 - LinkedHashMap
 - Caffeine
 - ◆ 远程缓存 (remote)
 - Redis
 - Tair

缓存

- SpringBoot提供了缓存的统一整合接口，方便缓存技术的开发与管理
 - ◆ Generic
 - ◆ JCache
 - ◆ Ehcache
 - ◆ Hazelcast
 - ◆ Infinispan
 - ◆ Couchbase
 - ◆ Redis
 - ◆ Caffeine
 - ◆ Simple (默认)
 - ◆ memcached
 - ◆ jcache (阿里)

缓存供应商变更:jetcache

- jetCache对SpringCache进行了封装，在原有功能基础上实现了多级缓存、缓存统计、自动刷新、异步调用、数据报表等功能
- jetCache设定了本地缓存与远程缓存的多级缓存解决方案
 - ◆ 本地缓存 (local)
 - LinkedHashMap
 - Caffeine
 - ◆ 远程缓存 (remote)
 - Redis
 - Tair

缓存供应商变更:jetcache

- 加入jetcache坐标

```
<dependency>  
  <groupId>com.alicp.jetcache</groupId>  
  <artifactId>jetcache-starter-redis</artifactId>  
  <version>2.6.2</version>  
</dependency>
```

缓存供应商变更:jetcache

- 配置**远程**缓存必要属性

```
jetcache:  
  remote:  
    default:  
      type: redis  
      host: localhost  
      port: 6379  
      poolConfig:  
        maxTotal: 50
```

缓存供应商变更:jetcache

- 配置**远程**缓存必要属性

```
jetcache:
  remote:
    default:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50
    sms:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50
```

缓存供应商变更:jetcache

- 配置**本地**缓存必要属性

```
jetcache:  
  local:  
    default:  
      type: linkedhashmap  
      keyConvertor: fastjson
```


缓存供应商变更:jetcache

- 配置范例

```
jetcache:  
  statIntervalMinutes: 15  
  areaInCacheName: false  
  local:  
    default:  
      type: linkedhashmap  
      keyConvertor: fastjson  
      limit: 100  
  remote:  
    default:  
      host: localhost  
      port: 6379  
      type: redis  
      keyConvertor: fastjson  
      valueEncoder: java  
      valueDecoder: java  
      poolConfig:  
        minIdle: 5  
        maxIdle: 20  
        maxTotal: 50
```

缓存供应商变更:jetcache

- 配置属性说明

属性	默认值	说明
jetcache.statIntervalMinutes	0	统计间隔，0表示不统计
jetcache.hiddenPackages	无	自动生成name时，隐藏指定的包名前缀
jetcache.[local remote].\${area}.type	无	缓存类型，本地支持linkedhashmap、caffeine，远程支持redis、tair
jetcache.[local remote].\${area}.keyConvertor	无	key转换器，当前仅支持fastjson
jetcache.[local remote].\${area}.valueEncoder	java	仅remote类型的缓存需要指定，可选java和kryo
jetcache.[local remote].\${area}.valueDecoder	java	仅remote类型的缓存需要指定，可选java和kryo
jetcache.[local remote].\${area}.limit	100	仅local类型的缓存需要指定，缓存实例最大元素数
jetcache.[local remote].\${area}.expireAfterWriteInMillis	无穷大	默认过期时间，毫秒单位
jetcache.local.\${area}.expireAfterAccessInMillis	0	仅local类型的缓存有效，毫秒单位，最大不活动间隔

缓存供应商变更:jetcache

- 开启jetcache注解支持

```
@SpringBootApplication
@EnableCreateCacheAnnotation
public class Springboot19CacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot19CacheApplication.class, args);
    }
}
```

缓存供应商变更:jetcache

- 声明缓存对象

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Autowired
    private CodeUtils codeUtils;
    @CreateCache(name = "smsCache", expire = 3600)
    private Cache<String, String> jetSMSCache;
}
```

缓存供应商变更:jcache

- 操作缓存

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Override
    public String sendCodeToSMS(String tele) {
        String code = this.codeUtils.generator(tele);
        jetSMSCache.put(tele,code);
        return code;
    }
    @Override
    public boolean checkCode(CodeMsg codeMsg) {
        String value = jetSMSCache.get(codeMsg.getTele());
        return codeMsg.getCode().equals(value);
    }
}
```



小结

1. jetcache简介
2. jetcache远程缓存使用方式
3. jetcache本地缓存使用方式

缓存供应商变更:jcache

- 启用方法注解

```
@SpringBootApplication
@EnableCreateCacheAnnotation
@EnableMethodCache(basePackages = "com.itheima")
public class Springboot20JetCacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot20JetCacheApplication.class, args);
    }
}
```

缓存供应商变更:jetcache

- 使用方法注解操作缓存

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;

    @Cached(name = "smsCache_", key = "#id", expire = 3600)
    @CacheRefresh(refresh = 10,timeUnit = TimeUnit.SECONDS)
    public Book getById(Integer id) {
        return bookDao.selectById(id);
    }
}
```


缓存供应商变更:jetcache

- 使用方法注解操作缓存

```
@Service
public class BookServiceImpl implements BookService {

    @CacheUpdate(name = "smsCache_", key = "#book.id", value = "#book")
    public boolean update(Book book) {
        return bookDao.updateById(book) > 0;
    }

    @CacheInvalidate(name = "smsCache_", key = "#id")
    public boolean delete(Integer id) {
        return bookDao.deleteById(id) > 0;
    }
}
```

缓存供应商变更:jcache

- 缓存对象必须保障可序列化

```
@Data
public class Book implements Serializable {
}
```

```
jetcache:
  remote:
    default:
      type: redis
      keyConvertor: fastjson
      valueEncoder: java
      valueDecoder: java
```

缓存供应商变更:jetcache

- 查看缓存统计报告

```
jetcache:
```

```
  statIntervalMinutes: 15
```



小结

1. jetcache方法注解使用方式

缓存供应商变更:j2cache

- j2cache是一个缓存整合框架，可以提供缓存的整合方案，使各种缓存搭配使用，自身不提供缓存功能
- 基于 ehcache + redis 进行整合

缓存供应商变更:j2cache

- 加入j2cache坐标，加入整合缓存的坐标

```
<dependency>
  <groupId>net.oschina.j2cache</groupId>
  <artifactId>j2cache-spring-boot2-starter</artifactId>
  <version>2.8.0-release</version>
</dependency>
<dependency>
  <groupId>net.oschina.j2cache</groupId>
  <artifactId>j2cache-core</artifactId>
  <version>2.8.4-release</version>
</dependency>
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
</dependency>
```

缓存供应商变更:j2cache

- 配置使用j2cache (application.yml)

```
j2cache:  
  config-location: j2cache.properties
```

缓存供应商变更:j2cache

- 配置一级缓存与二级缓存以及一级缓存数据到二级缓存的发送方式 (j2cache.properties)

```
# 配置1级缓存
j2cache.L1.provider_class = ehcache
ehcache.configXml = ehcache.xml

# 配置1级缓存数据到2级缓存的广播方式: 可以使用redis提供的消息订阅模式, 也可以使用jgroups多播实现
j2cache.broadcast = net.oschina.j2cache.cache.support.redis.SpringRedisPubSubPolicy

# 配置2级缓存
j2cache.L2.provider_class = net.oschina.j2cache.cache.support.redis.SpringRedisProvider
j2cache.L2.config_section = redis
redis.hosts = localhost:6379
```


缓存供应商变更:j2cache

- 设置使用缓存

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Autowired
    private CodeUtils codeUtils;
    @Autowired
    private CacheChannel cacheChannel;
}
```

缓存供应商变更:j2cache

- 设置使用缓存

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Override
    public String sendCodeToSMS(String tele) {
        String code = codeUtils.generator(tele);
        cacheChannel.set("sms",tele,code);
        return code;
    }
    @Override
    public boolean checkCode(SMSCode smsCode) {
        String code = cacheChannel.get("sms",smsCode.getTele()).asString();
        return smsCode.getCode().equals(code);
    }
}
```



小结

1. j2cache缓存的基础使用



总结

1. spring-cache

- simple
- ehcache
- redis
- memcached

2. jetcache

3. j2cache

任务

- 定时任务是企业级应用中的常见操作
 - ◆ 年度报表
 - ◆ 缓存统计报告
 - ◆
- 市面上流行的定时任务技术
 - ◆ Quartz
 - ◆ Spring Task

SpringBoot整合Quartz

- 相关概念
 - ◆ 工作 (Job) : 用于定义具体执行的工作
 - ◆ 工作明细 (JobDetail) : 用于描述定时工作相关的信息
 - ◆ 触发器 (Trigger) : 用于描述触发工作的规则, 通常使用cron表达式定义调度规则
 - ◆ 调度器 (Scheduler) : 描述了工作明细与触发器的对应关系

SpringBoot整合Quartz

- 导入SpringBoot整合quartz的坐标

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

SpringBoot整合Quartz

- 定义具体要执行的任务，继承QuartzJobBean

```
public class QuartzTaskBean extends QuartzJobBean {  
    @Override  
    protected void executeInternal(JobExecutionContext context) throws JobExecutionException {  
        System.out.println( "quartz job run... " );  
    }  
}
```


SpringBoot整合Quartz

- 定义工作明细与触发器，并绑定对应关系

```
@Configuration
public class QuartzConfig {
    @Bean
    public JobDetail printJobDetail(){
        return JobBuilder.newJob(QuartzTaskBean.class).storeDurably().build();
    }
    @Bean
    public Trigger printJobTrigger() {
        CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.cronSchedule("0/3 * * * * ?");
        return TriggerBuilder.newTrigger().forJob(printJobDetail())
            .withSchedule(cronScheduleBuilder).build();
    }
}
```



小结

1. SpringBoot整合Quartz

- 工作 (Job)
- 工作明细 (JobDetail)
- 触发器 (Trigger)
- 调度器 (Scheduler)

Spring Task

- 开启定时任务功能

```
@SpringBootApplication
@EnableScheduling
public class Springboot22TaskApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot22TaskApplication.class, args);
    }
}
```

Spring Task

- 设置定时执行的任务，并设定执行周期

```
@Component
public class ScheduledBean {
    @Scheduled(cron = "0/5 * * * * ?")
    public void printLog(){
        System.out.println(Thread.currentThread().getName()+":run...");
    }
}
```

Spring Task

- 定时任务相关配置

```
spring:
  task:
    scheduling:
      # 任务调度线程池大小 默认 1
    pool:
      size: 1
      # 调度线程名称前缀 默认 scheduling-
      thread-name-prefix: ssm_
    shutdown:
      # 线程池关闭时等待所有任务完成
      await-termination: false
      # 调度线程关闭前最大等待时间, 确保最后一定关闭
      await-termination-period: 10s
```



小结

1. Spring Task

- `@EnableScheduling`
- `@Scheduled`

SpringBoot整合JavaMail

- SMTP (Simple Mail Transfer Protocol) : 简单邮件传输协议, 用于**发送**电子邮件的传输协议
- POP3 (Post Office Protocol - Version 3) : 用于**接收**电子邮件的标准协议
- IMAP (Internet Mail Access Protocol) : 互联网消息协议, 是POP3的替代协议

SpringBoot整合JavaMail

- 导入SpringBoot整合JavaMail的坐标

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


SpringBoot整合JavaMail

- 配置JavaMail

```
spring:
  mail:
    host: smtp.qq.com
    username: *****@qq.com
    password: *****
```

POP3/IMAP/SMTP/Exchange/CardDAV/CalDAV服务

开启服务:	POP3/SMTP服务 (如何使用 Foxmail 等软件收发邮件?)	已关闭 开启
	IMAP/SMTP服务 (什么是 IMAP, 它又是如何设置?)	已关闭 开启
	Exchange服务 (什么是Exchange, 它又是如何设置?)	已关闭 开启
	CardDAV/CalDAV服务 (什么是CardDAV/CalDAV, 它又是如何设置?)	已关闭 开启
	(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置?)	

SpringBoot整合JavaMail

- 开启定时任务功能

```
@Service
public class SendMailServiceImpl implements SendMailService {
    private String from = "*****@qq.com"; // 发送人
    private String to = "*****@126.com"; // 接收人
    private String subject = "测试邮件"; // 邮件主题
    private String text = "测试邮件正文"; // 邮件内容
}
```

SpringBoot整合JavaMail

- 开启定时任务功能

```
@Service
public class SendMailServiceImpl implements SendMailService {
    @Autowired
    private JavaMailSender javaMailSender;
    @Override
    public void sendMail() {
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setFrom(from);
        mailMessage.setTo(to);
        mailMessage.setSubject(subject);
        mailMessage.setText(text);
        javaMailSender.send(mailMessage);
    }
}
```



小结

1. SpringBoot整合JavaMail发送简单邮件

SpringBoot整合JavaMail

- 附件与HTML文本支持

```
private String text = "<a href='https://www.itcast.cn/'>传智教育</a>";
@Override
public void sendMail() {
    try {
        MimeMessage mimeMessage = javaMailSender.createMimeMessage();
        MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMessage, true);
        mimeMessageHelper.setFrom(from);
        mimeMessageHelper.setTo(to);
        mimeMessageHelper.setSubject(subject);
        mimeMessageHelper.setText(text, true);
        File file = new File("logo.png");
        mimeMessageHelper.addAttachment("美图.png", file);
        javaMailSender.send(mimeMessage);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



小结

1. SpringBoot整合JavaMail发送邮件技巧

消息



消息

- 消息发送方
 - ◆ 生产者
- 消息接收方
 - ◆ 消费者

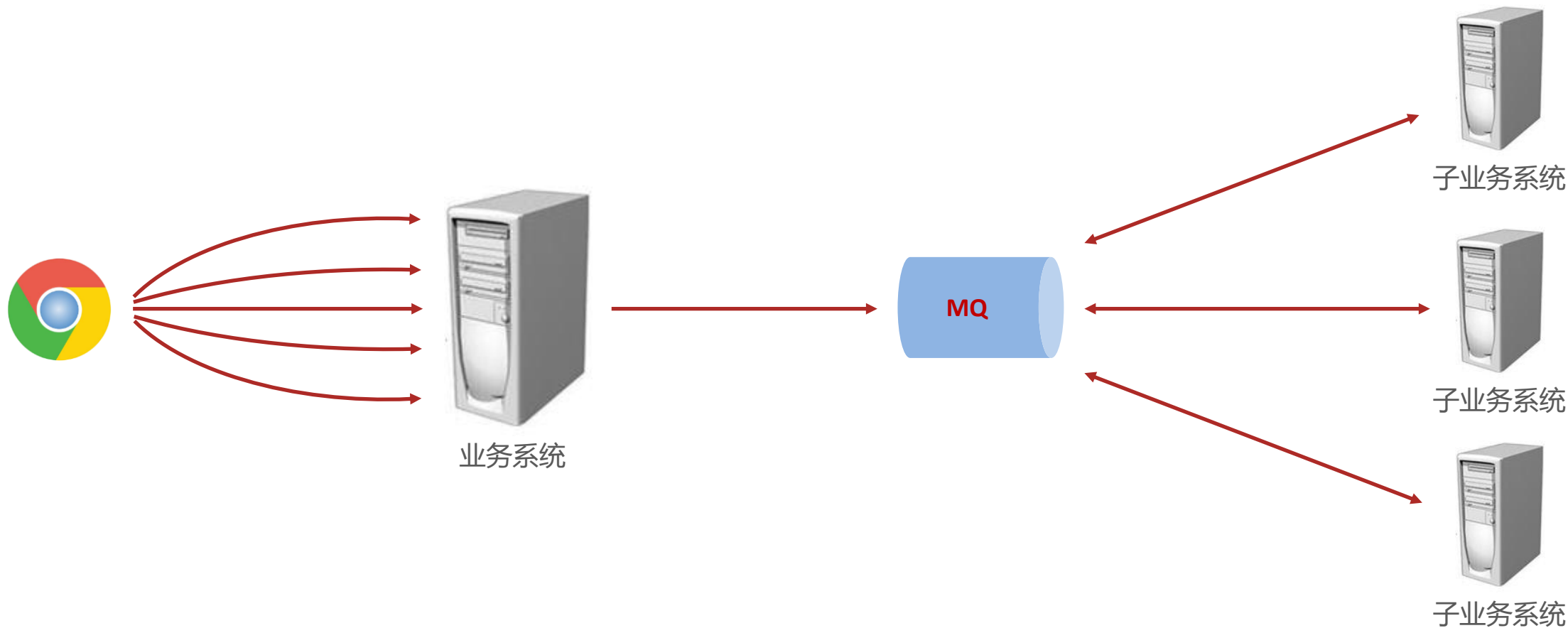


消息

- 同步消息
- 异步消息



消息



消息

- 企业级应用中广泛使用的三种异步消息传递技术
 - ◆ JMS
 - ◆ AMQP
 - ◆ MQTT

JMS

- JMS (Java Message Service) : 一个规范，等同于JDBC规范，提供了与消息服务相关的API接口
- JMS消息模型
 - ◆ peer-2-peer: 点对点模型，消息发送到一个队列中，队列保存消息。队列的消息只能被一个消费者消费，或超时
 - ◆ **publish-subscribe**: 发布订阅模型，消息可以被多个消费者消费，生产者和消费者完全独立，不需要感知对方的存在
- JMS消息种类
 - ◆ TextMessage
 - ◆ MapMessage
 - ◆ **BytesMessage**
 - ◆ StreamMessage
 - ◆ ObjectMessage
 - ◆ Message (只有消息头和属性)
- JMS实现: ActiveMQ、Redis、HornetMQ、RabbitMQ、RocketMQ (没有完全遵守JMS规范)

AMQP

- AMQP (advanced message queuing protocol)：一种协议（高级消息队列协议，也是消息代理规范），规范了网络交换的数据格式，兼容JMS
- 优点：具有跨平台性，服务器供应商，生产者，消费者可以使用不同的语言来实现
- AMQP消息模型
 - ◆ direct exchange
 - ◆ fanout exchange
 - ◆ topic exchange
 - ◆ headers exchange
 - ◆ system exchange
- AMQP消息种类：byte[]
- AMQP实现：RabbitMQ、StormMQ、RocketMQ

MQTT

- MQTT (Message Queueing Telemetry Transport) 消息队列遥测传输，专为小设备设计，是物联网 (IOT) 生态系统中主要成分之一

Kafka

- Kafka, 一种高吞吐量的分布式发布订阅消息系统, 提供实时消息功能。

消息

- ActiveMQ
- RabbitMQ
- RocketMQ
- Kafka



小结

1. 消息概念与作用
2. JMS
3. AMQP
4. MQTT

消息案例——订单短信通知

- 购物订单业务
 - ◆ 登录状态检测
 - ◆ 生成主单
 - ◆ 生成子单
 - ◆ 库存检测与变更
 - ◆ 积分变更
 - ◆ 支付
 - ◆ 短信通知
 - ◆ 购物车维护
 - ◆ 运单信息初始化
 - ◆ 商品库存维护
 - ◆ 会员维护
 - ◆ ...

消息案例——订单短信通知

- 购物订单业务
 - ◆ 登录状态检测
 - ◆ 生成主单
 - ◆ 生成子单
 - ◆ 库存检测与变更
 - ◆ 积分变更
 - ◆ 支付
 - ◆ 短信通知（异步）
 - ◆ 购物车维护
 - ◆ 运单信息初始化
 - ◆ 商品库存维护
 - ◆ 会员维护
 - ◆ ...



小结

1. 订单短信通知案例

ActiveMQ

- 下载地址: <https://activemq.apache.org/components/classic/download/>
- 安装: 解压缩

ActiveMQ

- 启动服务

```
activemq.bat
```

- 访问服务器

```
http://127.0.0.1:8161/
```

- ◆ 服务端口: 61616, 管理后台端口: 8161
- ◆ 用户名&密码: **admin**



小结

1. ActiveMQ下载与安装
2. ActiveMQ服务启动（控制台）

SpringBoot整合ActiveMQ

- 导入SpringBoot整合ActiveMQ坐标

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


SpringBoot整合ActiveMQ

- 配置ActiveMQ (采用默认配置)

```
spring:  
  activemq:  
    broker-url: tcp://localhost:61616  
  jms:  
    pub-sub-domain: true  
  template:  
    default-destination: itheima
```

SpringBoot整合ActiveMQ

- 生产与消费消息（使用默认消息存储队列）

```
@Service
public class MessageServiceActivemqImpl implements MessageService {
    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;

    public void sendMessage(String id) {
        System.out.println("使用Active将待发送短信的订单纳入处理队列, id: "+id);
        jmsMessagingTemplate.convertAndSend(id);
    }
    public String doMessage() {
        return jmsMessagingTemplate.receiveAndConvert(String.class);
    }
}
```

SpringBoot整合ActiveMQ

- 生产与消费消息（指定消息存储队列）

```
@Service
public class MessageServiceActivemqImpl implements MessageService {
    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;

    public void sendMessage(String id) {
        System.out.println("使用Active将待发送短信的订单纳入处理队列, id: "+id);
        jmsMessagingTemplate.convertAndSend("order.sm.queue.id",id);
    }
    public String doMessage() {
        return jmsMessagingTemplate.receiveAndConvert("order.sm.queue.id",String.class);
    }
}
```

SpringBoot整合ActiveMQ

- 使用消息监听器对消息队列监听

```
@Component
public class MessageListener {
    @JmsListener(destination = "order.sm.queue.id")
    public void receive(String id){
        System.out.println("已完成短信发送业务, id: "+id);
    }
}
```

SpringBoot整合ActiveMQ

- 流程性业务消息消费完转入下一个消息队列

```
@Component
public class MessageListener {
    @JmsListener(destination = "order.sm.queue.id")
    @SendTo("order.other.queue.id")
    public String receive(String id){
        System.out.println("已完成短信发送业务, id: "+id);
        return "new:"+id;
    }
}
```

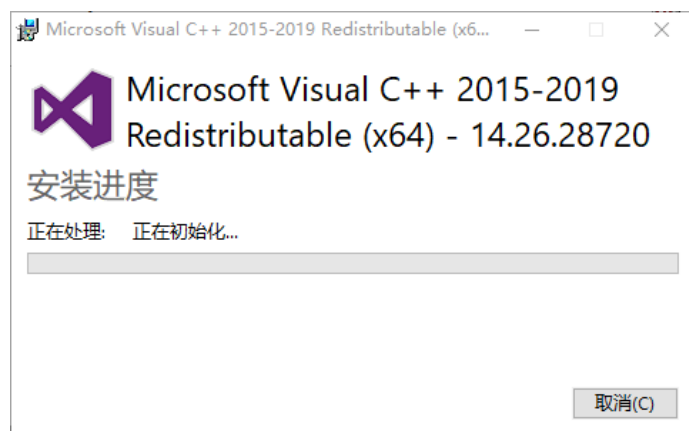


小结

1. SpringBoot整合ActiveMQ

RabbitMQ

- RabbitMQ基于Erlang语言编写，需要安装Erlang
- Erlang
 - ◆ 下载地址: <https://www.erlang.org/downloads>
 - ◆ 安装: 一键傻瓜式安装，安装完毕需要重启，需要依赖Windows组件
 - ◆ 环境变量配置
 - ERLANG_HOME
 - PATH



RabbitMQ

- 下载地址: <https://rabbitmq.com/install-windows.html>
- 安装: 一键傻瓜式安装

RabbitMQ

- 启动服务

```
rabbitmq-service.bat start
```

- 关闭服务

```
rabbitmq-service.bat stop
```

- 查看服务状态

```
rabbitmqctl status
```

RabbitMQ

- 服务管理可视化（插件形式）
- 查看已安装的插件列表

```
rabbitmq-plugins.bat list
```

- 开启服务管理插件

```
rabbitmq-plugins.bat enable rabbitmq_management
```

- 访问服务器

```
http://localhost:15672
```

- ◆ 服务端口：5672，管理后台端口：15672
- ◆ 用户名&密码：**guest**



小结

1. Erlang下载与安装 (环境变量配置)
2. RabbitMQ下载与安装
3. RabbitMQ服务启动 (服务)
4. RabbitMQ服务管理

JMS

- JMS (Java Message Service) : 一个规范，等同于JDBC规范，提供了与消息服务相关的API接口
- JMS消息模型
 - ◆ peer-2-peer: 点对点模型，消息发送到一个队列中，队列保存消息。队列的消息只能被一个消费者消费，或超时
 - ◆ **publish-subscribe**: 发布订阅模型，消息可以被多个消费者消费，生产者和消费者完全独立，不需要感知对方的存在
- JMS消息种类
 - ◆ TextMessage
 - ◆ MapMessage
 - ◆ **BytesMessage**
 - ◆ StreamMessage
 - ◆ ObjectMessage
 - ◆ Message (只有消息头和属性)
- JMS实现: ActiveMQ、Redis、HornetMQ、RabbitMQ、RocketMQ (没有完全遵守JMS规范)

AMQP

- AMQP (advanced message queuing protocol) : 一种协议 (高级消息队列协议, 也是消息代理规范), 规范了网络交换的数据格式, 兼容JMS
- 优点: 具有跨平台性, 服务器供应商, 生产者, 消费者可以使用不同的语言来实现
- AMQP消息模型
 - ◆ direct exchange
 - ◆ fanout exchange
 - ◆ topic exchange
 - ◆ headers exchange
 - ◆ system exchange
- AMQP消息种类: byte[]
- AMQP实现: RabbitMQ、StormMQ、RocketMQ

SpringBoot整合RabbitMQ

- 导入SpringBoot整合RabbitMQ坐标

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

SpringBoot整合RabbitMQ

- 配置RabbitMQ (采用默认配置)

```
spring:  
  rabbitmq:  
    host: localhost  
    port: 5672
```

SpringBoot整合RabbitMQ

- 定义消息队列(direct)

```
@Configuration
public class RabbitDirectConfig {
    @Bean
    public Queue queue(){
        return new Queue("simple_queue");
    }
}
```


SpringBoot整合RabbitMQ

- 定义消息队列(direct)

```
@Configuration
public class RabbitDirectConfig {
    @Bean
    public Queue queue(){
        // durable:是否持久化,默认false
        // exclusive:是否当前连接专用,默认false,连接关闭后队列即被删除
        // autoDelete:是否自动删除,当生产者或消费者不再使用此队列,自动删除
        return new Queue("simple_queue",true,false,false);
    }
}
```

SpringBoot整合RabbitMQ

- 定义消息队列(direct)

```
@Configuration
public class RabbitDirectConfig {
    @Bean
    public Queue directQueue(){        return new Queue("direct_queue");    }
    @Bean
    public Queue directQueue2(){        return new Queue("direct_queue2");    }
    @Bean
    public DirectExchange directExchange(){
        return new DirectExchange("directExchange");
    }
    @Bean
    public Binding bindingDirect(){
        return BindingBuilder.bind(directQueue()).to(directExchange()).with("direct");
    }
    @Bean
    public Binding bindingDirect2(){
        return BindingBuilder.bind(directQueue2()).to(directExchange()).with("direct2");
    }
}
```

SpringBoot整合RabbitMQ

- 生产与消费消息(direct)

```
@Service
public class MessageServiceRabbitmqDirectImpl implements MessageService {
    @Autowired
    private AmqpTemplate amqpTemplate;
    @Override
    public void sendMessage(String id) {
        System.out.println("使用Rabbitmq将待发送短信的订单纳入处理队列, id: "+id);
        amqpTemplate.convertAndSend("directExchange","direct",id);
    }
}
```

SpringBoot整合RabbitMQ

- 使用消息监听器对消息队列监听(direct)

```
@Component
public class RabbitMessageListener {
    @RabbitListener(queues = "direct_queue")
    public void receive(String id){
        System.out.println("已完成短信发送业务, id: "+id);
    }
}
```

SpringBoot整合RabbitMQ

- 使用多消息监听器对消息队列监听进行消息轮循处理(direct)

```
@Component
public class RabbitMessageListener2 {
    @RabbitListener(queues = "direct_queue")
    public void receive(String id){
        System.out.println("已完成短信发送业务 (two) , id: "+id);
    }
}
```



小结

1. SpringBoot整合RabbitMQ直连交换机模式

SpringBoot整合RabbitMQ

- 定义消息队列(topic)

```
@Configuration
public class RabbitTopicConfig {
    @Bean
    public Queue topicQueue(){        return new Queue("topic_queue");    }
    @Bean
    public Queue topicQueue2(){        return new Queue("topic_queue2");    }
    @Bean
    public TopicExchange topicExchange(){
        return new TopicExchange("topicExchange");
    }
    @Bean
    public Binding bindingTopic(){
        return BindingBuilder.bind(topicQueue()).to(topicExchange()).with("topic.*.*");
    }
    @Bean
    public Binding bindingTopic2(){
        return BindingBuilder.bind(topicQueue2()).to(topicExchange()).with("topic.#");
    }
}
```

SpringBoot整合RabbitMQ

- 绑定键匹配规则
 - ◆ * (星号): 用来表示一个单词，且该单词是必须出现的
 - ◆ # (井号): 用来表示任意数量

匹配键	topic.*.*	topic.#
topic.order.id	true	true
order.topic.id	false	false
topic.sm.order.id	false	true
topic.sm.id	false	true
topic.id.order	true	true
topic.id	false	true
topic.order	false	true

SpringBoot整合RabbitMQ

- 生产与消费消息(topic)

```
@Service
public class MessageServiceRabbitmqTopicmpl implements MessageService {
    @Autowired
    private AmqpTemplate amqpTemplate;
    @Override
    public void sendMessage(String id) {
        System.out.println("使用Rabbitmq将待发送短信的订单纳入处理队列, id: "+id);
        amqpTemplate.convertAndSend("topicExchange","topic.order.id",id);
    }
}
```

SpringBoot整合RabbitMQ

- 使用消息监听器对消息队列监听(topic)

```
@Component
public class RabbitTopicMessageListener {
    @RabbitListener(queues = "topic_queue")
    public void receive(String id){
        System.out.println("已完成短信发送业务, id: "+id);
    }
    @RabbitListener(queues = "topic_queue2")
    public void receive2(String id){
        System.out.println("已完成短信发送业务(two), id: "+id);
    }
}
```



小结

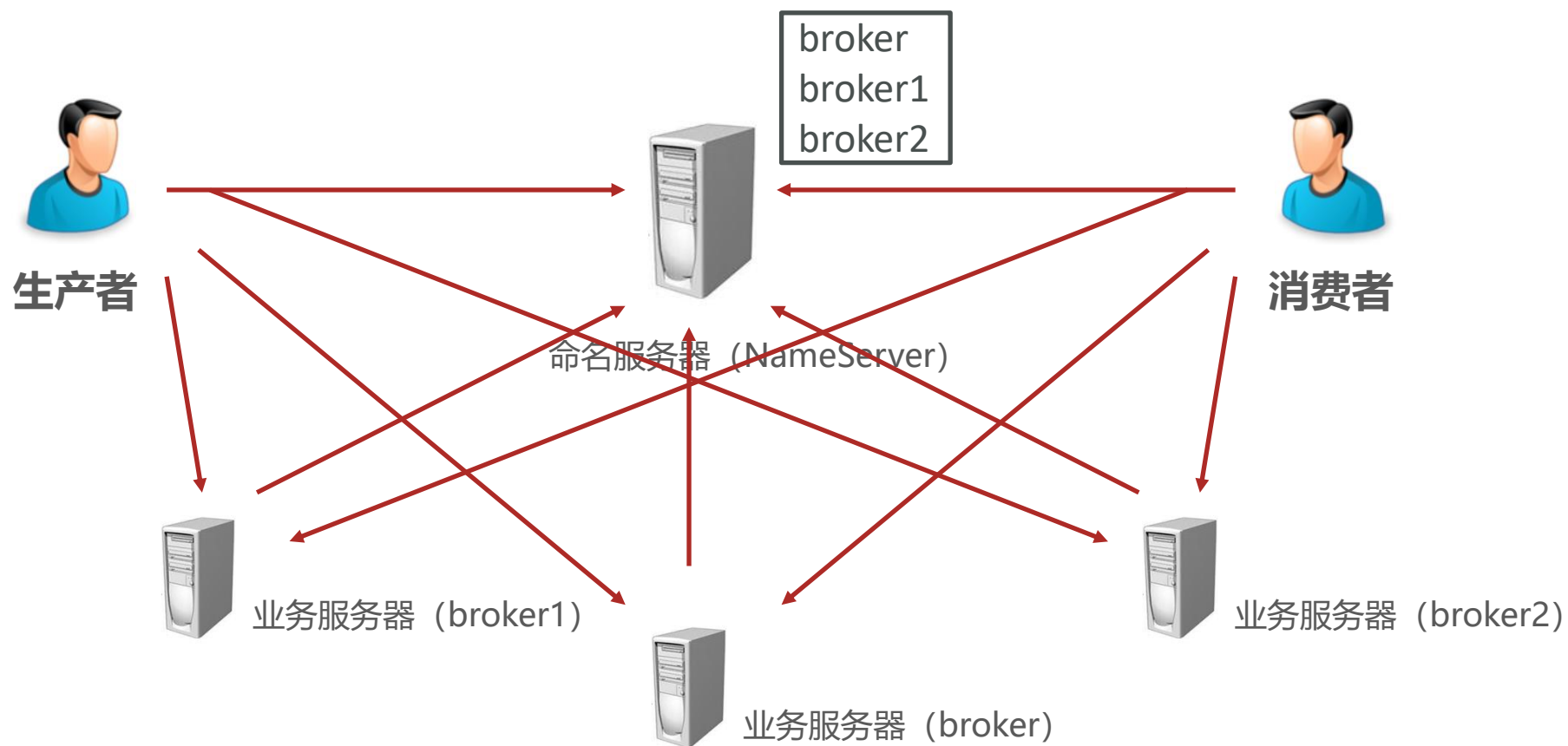
1. SpringBoot整合RabbitMQ主题交换机模式

RocketMQ

- 下载地址: <https://rocketmq.apache.org/>
- 安装: 解压缩
 - ◆ 默认服务端口: 9876
- 环境变量配置
 - ◆ ROCKETMQ_HOME
 - ◆ PATH
 - ◆ NAMESRV_ADDR (建议) : 127.0.0.1:9876

RocketMQ

- 命名服务器与broker



RocketMQ

- 启动命名服务器

```
mqnamesrv
```

- 启动broker

```
mqbroker
```

RocketMQ

- 服务器功能测试：生产者

```
tools org.apache.rocketmq.example.quickstart.Producer
```

- 服务器功能测试：消费者

```
tools org.apache.rocketmq.example.quickstart.Consumer
```



小结

1. RocketMQ下载与安装 (环境变量配置)
2. 命名服务器启动 (控制台)
3. broker服务启动 (控制台)
4. 消息生产消费测试

消息

- 导入SpringBoot整合RocketMQ坐标

```
<dependency>
  <groupId>org.apache.rocketmq</groupId>
  <artifactId>rocketmq-spring-boot-starter</artifactId>
  <version>2.2.1</version>
</dependency>
```

消息

- 配置RocketMQ（采用默认配置）

```
rocketmq:  
  name-server: localhost:9876  
  producer:  
    group: group_rocketmq
```

消息

- 生产消息

```
@Service
public class MessageServiceRocketmqImpl implements MessageService {
    @Autowired
    private RocketMQTemplate rocketMQTemplate;
    @Override
    public void sendMessage(String id) {
        rocketMQTemplate.convertAndSend("order_sm_id",id);
        System.out.println("使用Rabbitmq将待发送短信的订单纳入处理队列, id: "+id);
    }
}
```

消息

- 生产异步消息

```
@Service
public class MessageServiceRocketmqImpl implements MessageService {
    @Autowired
    private RocketMQTemplate rocketMQTemplate;
    @Override
    public void sendMessage(String id) {
        SendCallback callback = new SendCallback() {
            @Override
            public void onSuccess(SendResult sendResult) {
                System.out.println("消息发送成功");
            }

            @Override
            public void onException(Throwable throwable) {
                System.out.println("消息发送失败!!!!!!!!!!!!");
            }
        };
        System.out.println("使用Rabbitmq将待发送短信的订单纳入处理队列, id: "+id);
        rocketMQTemplate.asyncSend("order_sm_id",id,callback);
    }
}
```

消息

- 使用消息监听器对消息队列监听

```
@Component
@RocketMQMessageListener(topic="order_sm_id",consumerGroup = "group_rocketmq")
public class RocketmqMessageListener implements RocketMQListener<String> {
    @Override
    public void onMessage(String id) {
        System.out.println("已完成短信发送业务, id: "+id);
    }
}
```



小结

1. SpringBoot整合RocketMQ

Kafka

- 下载地址: <https://kafka.apache.org/downloads>
 - ◆ windows 系统下3.0.0版本存在BUG, 建议使用2.X版本
- 安装: 解压缩

Kafka

- 启动zookeeper

```
zookeeper-server-start.bat ..\..\config\zookeeper.properties
```

◆ 默认端口: 2181

- 启动kafka

```
kafka-server-start.bat ..\..\config\server.properties
```

◆ 默认端口: 9092

Kafka

- 创建topic

```
kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --  
partitions 1 --topic itheima
```

- 查看topic

```
kafka-topics.bat --zookeeper 127.0.0.1:2181 --list
```

- 删除topic

```
kafka-topics.bat --delete --zookeeper localhost:2181 --topic itheima
```

Kafka

- 生产者功能测试

```
kafka-console-producer.bat --broker-list localhost:9092 --topic itheima
```

- 消费者功能测试

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic itheima --from-  
beginning
```



小结

1. Kafka下载与安装（环境变量配置）
2. zookeeper启动（控制台）
3. kafka服务启动（控制台）
4. topic维护
5. 消息生产消费测试

消息

- 导入SpringBoot整合Kafka坐标

```
<dependency>  
  <groupId>org.springframework.kafka</groupId>  
  <artifactId>spring-kafka</artifactId>  
</dependency>
```

消息

- 配置Kafka (采用默认配置)

```
spring:  
  kafka:  
    bootstrap-servers: localhost:9092  
    consumer:  
      group-id: order
```

消息

- 生产消息

```
@Service
public class MessageServiceKafkaImpl implements MessageService {
    @Autowired
    private KafkaTemplate<String ,String> kafkaTemplate;
    @Override
    public void sendMessage(String id) {
        System.out.println("使用Kafka将待发送短信的订单纳入处理队列, id: "+id);
        kafkaTemplate.send("kafka_topic",id);
    }
}
```

消息

- 使用消息监听器对消息队列监听

```
@Component
public class KafkaMessageListener{
    @KafkaListener(topics = {"kafka_topic"})
    public void onMessage(ConsumerRecord<?, ?> record) {
        System.out.println("已完成短信发送业务, id: "+record.value());
    }
}
```



小结

1. SpringBoot整合Kafka



总结

1. 消息
2. ActiveMQ
3. RabbitMQ
4. RocketMQ
5. Kafka



总结

1. 缓存
2. 任务
3. 邮件
4. 消息



监控

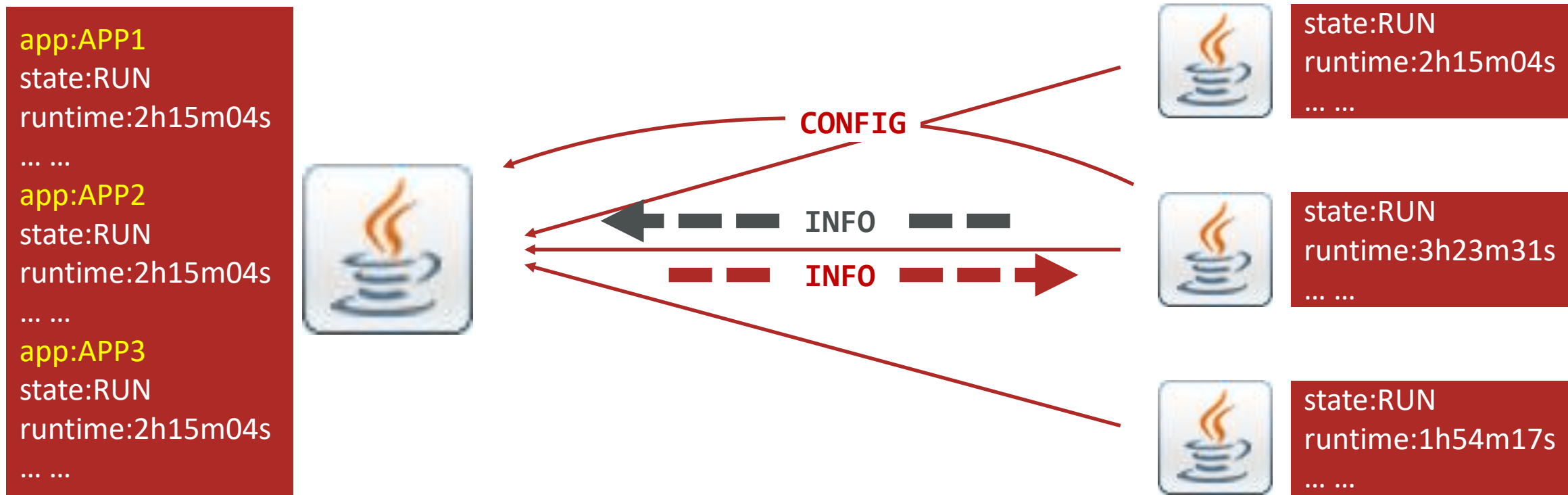
- 监控的意义
- 可视化监控平台
- 监控原理
- 自定义监控指标

监控的意义

- 监控服务状态是否宕机
- 监控服务运行指标（内存、虚拟机、线程、请求等）
- 监控日志
- 管理服务（服务下线）

监控的实施方式

- 显示监控信息的服务器：用于获取服务信息，并显示对应的信息
- 运行的服务：启动时主动上报，告知监控服务器自己需要受到监控





小结

1. 监控的意义
2. 监控方式

可视化监控平台

- Spring Boot Admin, 开源社区项目, 用于管理和监控SpringBoot应用程序。客户端注册到服务端后, 通过HTTP请求方式, 服务端定期从客户端获取对应的信息, 并通过UI界面展示对应信息。

可视化监控平台

- Admin服务端

```
<properties>
  <spring-boot-admin.version>2.5.4</spring-boot-admin.version>
</properties>
<dependencies>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-dependencies</artifactId>
      <version>${spring-boot-admin.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```


可视化监控平台

- Admin客户端

```
<properties>
  <spring-boot-admin.version>2.5.4</spring-boot-admin.version>
</properties>
<dependencies>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-dependencies</artifactId>
      <version>${spring-boot-admin.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

可视化监控平台

- Admin服务端

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-server</artifactId>
  <version>2.5.4</version>
</dependency>
```

- Admin客户端

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
  <version>2.5.4</version>
</dependency>
```

可视化监控平台

- Admin服务端

```
server:  
  port: 8080
```

- 设置启用Spring-Admin

```
@SpringBootApplication  
@EnableAdminServer  
public class Springboot25ActuatorServerApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(Springboot25ActuatorServerApplication.class, args);  
    }  
}
```

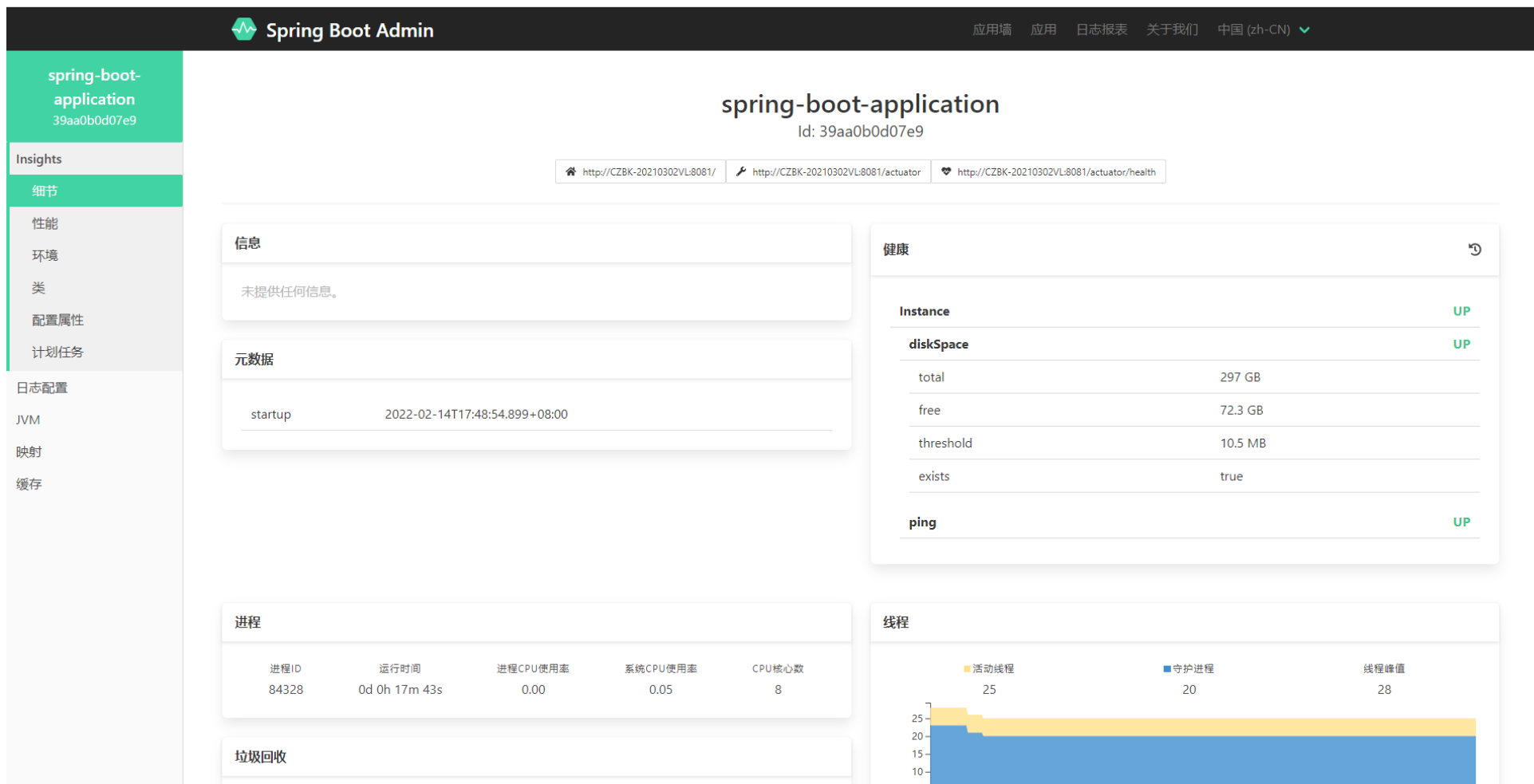
可视化监控平台

- Admin客户端

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:8080
management:
  endpoint:
    health:
      show-details: always
endpoints:
  web:
    exposure:
      include: "*"

```

可视化监控平台





小结

1. 可视化监控平台——Spring Boot Admin
2. 配置通过web端读取监控信息

监控原理

- Actuator提供了SpringBoot生产就绪功能，通过端点的配置与访问，获取端点信息
- 端点描述了一组监控信息，SpringBoot提供了多个内置端点，也可以根据需要自定义端点信息
- 访问当前应用所有端点信息：**/actuator**
- 访问端点详细信息：**/actuator/端点名称**

监控原理

ID	描述	默认启用
auditevents	暴露当前应用程序的审计事件信息。	是
beans	显示应用程序中所有 Spring bean 的完整列表。	是
caches	暴露可用的缓存。	是
conditions	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因。	是
configprops	显示所有 @ConfigurationProperties 的校对清单。	是
env	暴露 Spring ConfigurableEnvironment 中的属性。	是
flyway	显示已应用的 Flyway 数据库迁移。	是
health	显示应用程序健康信息	是
httptrace	显示 HTTP 追踪信息（默认情况下，最后 100 个 HTTP 请求/响应交换）。	是
info	显示应用程序信息。	是
integrationgraph	显示 Spring Integration 图。	是

监控原理

ID	描述	默认启用
loggers	显示和修改应用程序中日志记录器的配置。	是
liquibase	显示已应用的 Liquibase 数据库迁移。	是
metrics	显示当前应用程序的指标度量信息。	是
mappings	显示所有 @RequestMapping 路径的整理清单。	是
scheduledtasks	显示应用程序中的调度任务。	是
sessions	允许从 Spring Session 支持的会话存储中检索和删除用户会话。当使用 Spring Session 的响应式 Web 应用程序支持时不可用。	是
shutdown	正常关闭应用程序。	否
threaddump	执行线程 dump。	是

监控原理

- Web程序专用端点

ID	描述	默认启用
heapdump	返回一个 hprof 堆 dump 文件。	是
jolokia	通过 HTTP 暴露 JMX bean（当 Jolokia 在 classpath 上时，不适用于 WebFlux）。	是
logfile	返回日志文件的内容（如果已设置 logging.file 或 logging.path 属性）。支持使用 HTTP Range 头来检索部分日志文件的内容。	是
prometheus	以可以由 Prometheus 服务器抓取的格式暴露指标。	是

监控原理

- 启用指定端点

```
management:  
  endpoint:  
    health: # 端点名称  
      enabled: true  
      show-details: always  
    beans: # 端点名称  
      enabled: true
```

- 启用所有端点

```
management:  
  endpoints:  
    enabled-by-default: true
```

监控原理

- 暴露端点功能
 - ◆ 端点中包含的信息存在敏感信息，需要对外暴露端点功能时手动设定指定端点信息

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

监控原理

- 暴露端点功能

ID	JMX	Web
auditevents	是	否
beans	是	否
caches	是	否
conditions	是	否
configprops	是	否
env	是	否
flyway	是	否
health	是	是
heapdump	N/A	否
httptrace	是	否
info	是	是

监控原理

- 暴露端点功能

ID	JMX	Web
integrationgraph	是	否
jolokia	N/A	否
logfile	N/A	否
loggers	是	否
liquibase	是	否
metrics	是	否
mappings	是	否
prometheus	N/A	否
scheduledtasks	是	否
sessions	是	否
shutdown	是	否
threaddump	是	否



小结

1. Actuator
2. 端点功能开启与关闭
3. 端点功能暴露

自定义监控指标

- 为info端点添加自定义指标

```
info:  
  appName: @project.artifactId@  
  version: @project.version@  
  author: itheima
```


自定义监控指标

- 为info端点添加自定义指标

```
@Component
public class AppInfoContributor implements InfoContributor {
    @Override
    public void contribute(Info.Builder builder) {
        Map<String, Object> infoMap = new HashMap<>();
        infoMap.put("buildTime", "2006");
        builder.withDetail("runTime", System.currentTimeMillis())
                .withDetail("company", "传智教育");
        builder.withDetails(infoMap);
    }
}
```



小结

1. 自定义info端点信息

自定义监控指标

- 为Health端点添加自定义指标

```
@Component
public class AppHealthContributor extends AbstractHealthIndicator {
    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        boolean condition = true;
        if(condition){
            Map<String,Object> infoMap = new HashMap<>();
            infoMap.put("buildTime","2006");
            builder.withDetail("runTime",System.currentTimeMillis())
                    .withDetail("company","传智教育");
            builder.withDetails(infoMap);
            builder.status(Status.UP);
        }else{
            builder.status(Status.DOWN);
        }
    }
}
```



小结

1. 自定义Health端点信息

自定义监控指标

- 为Metrics端点添加自定义指标

```
@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book> implements IBookService {
    private Counter counter;
    public BookServiceImpl(MeterRegistry meterRegistry){
        counter = meterRegistry.counter("用户付费操作次数: ");
    }
    @Override
    public boolean delete(Integer id) {
        counter.increment();
        return bookDao.deleteById(id) > 0;
    }
}
```



小结

1. 自定义Metrics端点信息

自定义监控指标

- 自定义端点

```
@Component
@Endpoint(id="pay")
public class PayEndPoint {
    @ReadOperation
    public Object getPay(){
        //调用业务操作, 获取支付相关信息结果, 最终return出去
        Map payMap = new HashMap();
        payMap.put("level 1",103);
        payMap.put("level 2",315);
        payMap.put("level 3",666);
        return payMap;
    }
}
```



小结

1. 自定义端点



总结

1. 监控的意义
2. 可视化监控平台——Spring Boot Admin
3. 监控原理——Actuator
4. 自定义监控指标
 - 系统端点添加监控指标
 - 自定义端点



传智教育旗下高端IT教育品牌