

SpringBoot的高级教程

九、SpringBoot缓存

缓存的场景

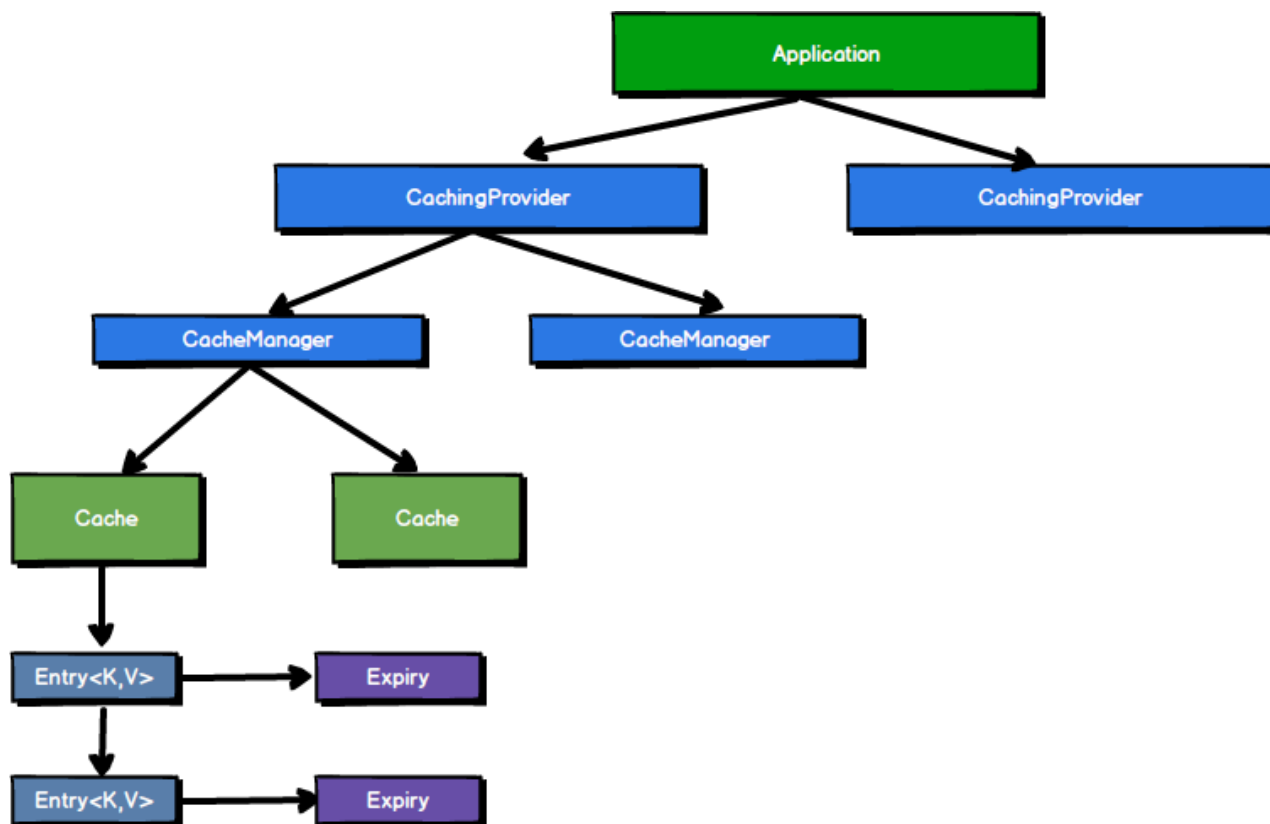
- 临时性数据存储【校验码】
- 避免频繁因为相同的内容查询数据库【查询的信息】

1、JSR107缓存规范

用的比较少

Java Caching定义了5个核心接口

- CachingProvider
定义了创建、配置、获取、管理和控制多个CacheManager。一个应用可以在运行期间访问多个CachingProvider
- CacheManager
定义了创建、配置、获取、管理和控制多个唯一命名的Cache,这些Cache存在于CacheManager的上下文中，一个CacheManager只被一个CachingProvider拥有
- Cache
类似于Map的数据结构并临时储存以key为索引的值，一个Cache仅仅被一个CacheManager所拥有
- Entry
存储在Cache中的key-value对
- Expiry
存储在Cache的条目有一个定义的有效期，一旦超过这个时间，就会设置过期的状态，过期无法被访问，更新，删除。缓存的有效期可以通过ExpiryPolicy设置。



2、Spring的缓存抽象

包括一些JSR107的注解

CachingManager

Cache

1、基本概念

重要的概念&缓存注解

	功能
Cache	缓存接口，定义缓存操作，实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存（Cache）组件
@Cacheable	针对方法配置，根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存 update，调用，将信息更新缓存
@EnableCaching	开启基于注解的缓存
KeyGenerator	缓存数据时key生成的策略
serialize	缓存数据时value序列化策略

2、整合项目

- 1、新建一个SpringBoot1.5+web+mysql+mybatis+cache
- 2、编写配置文件，连接Mysql

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/spring_cache
spring.datasource.username=root
spring.datasource.password=123456
# 开启驼峰命名匹配
mybatis.configuration.map-underscore-to-camel-case=true
# 打印sql
logging.level.com.cuzz.cache.mapper=debug
# 可以打印配置报告
debug=true
```

- 3、创建一个bean实例

Department

```
package com.cuzz.cache.bean;

import java.io.Serializable;

public class Department implements Serializable {

    private Integer id;
    private String deptName;

    public Department(){
    }

    // getter/setter
```

```
}
```

Employee

```
package com.cuzz.cache.bean;

import java.io.Serializable;

public class Employee implements Serializable {

    private Integer id;
    private String lastName;
    private String gender;
    private String email;
    private Integer dId;

    public Employee() {
    }
    // getter/setter
}
```

4、创建mapper接口映射数据库，并访问数据库中的数据

```
package com.cuzz.cache.mapper;

import com.cuzz.cache.bean.Employee;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

/**
 * @Author: cuzz
 * @Date: 2018/9/26 15:54
 * @Description:
 */
@Mapper
public interface EmployeeMapper {

    @Select("SELECT * FROM employee WHERE id = #{id}")
    Employee getEmployeeById(Integer id);

    @Update("UPDATE employee SET last_name=#{lastName},email=#{email},gender=#{gender},d_id=#{dId} WHERE id=#{id}")
    void updateEmp(Employee employee);

    @Delete("DELETE FROM employee WHERE employee.id=#{id}")
    void deleteEmp(Integer id);

    @Select("SELECT * FROM employee WHERE last_name=#{lastName}")
```

```
Employee getEmpByLastName(String lastName);

}
```

5、主程序添加注解MapperScan，并且使用@EnableCaching开启缓存

```
package com.cuzz.cache;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@MapperScan("com.cuzz.cache.mapper")
@SpringBootApplication
@EnableCaching
public class Springboot07CacheApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot07CacheApplication.class, args);
    }
}
```

6、编写service，来具体实现mapper中的方法

将方法的运行结果进行缓存，以后要是再有相同的数据，直接从缓存中获取，不用调用方法

CacheManager中管理多个Cache组件，对缓存的真正CRUD操作在Cache组件中，每个缓存组件都有自己的唯一名字；

属性：

- CacheName/value:指定存储缓存组件的名字
- key:缓存数据使用的key,可以使用它来指定。默认是使用方法参数的值，1-方法的返回值
- 编写Spel表达式: #id 参数id的值， #a0/#p0 #root.args[0]
- keyGenerator:key的生成器，自己可以指定key的生成器的组件id
- key/keyGendertor二选一使用
- cacheManager指定Cache管理器，或者cacheReslover指定获取解析器
- condition:指定符合条件的情况下，才缓存；
- unless: 否定缓存，unless指定的条件为true，方法的返回值就不会被缓存，可以获取到结果进行判断
- sync:是否使用异步模式，unless不支持

```
package com.cuzz.cache.service;

import com.cuzz.cache.bean.Employee;
import com.cuzz.cache.mapper.EmployeeMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
```

```

/**
 * @Author: cuzz
 * @Date: 2018/9/26 16:08
 * @Description:
 */
@Service
public class EmployeeService {

    @Autowired
    EmployeeMapper employeeMapper;

    @Cacheable(cacheNames = "emp")
    public Employee getEmployee(Integer id) {
        System.out.println("----> 查询" + id + "号员工");
        return employeeMapper.getEmployeeById(id);
    }
}

```

7、编写controller测试

```

@RestController
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @GetMapping("/emp/{id}")
    public Employee getEmp(@PathVariable("id") Integer id){
        return employeeService.getEmp(id);
    }
}

```

8、测试结果

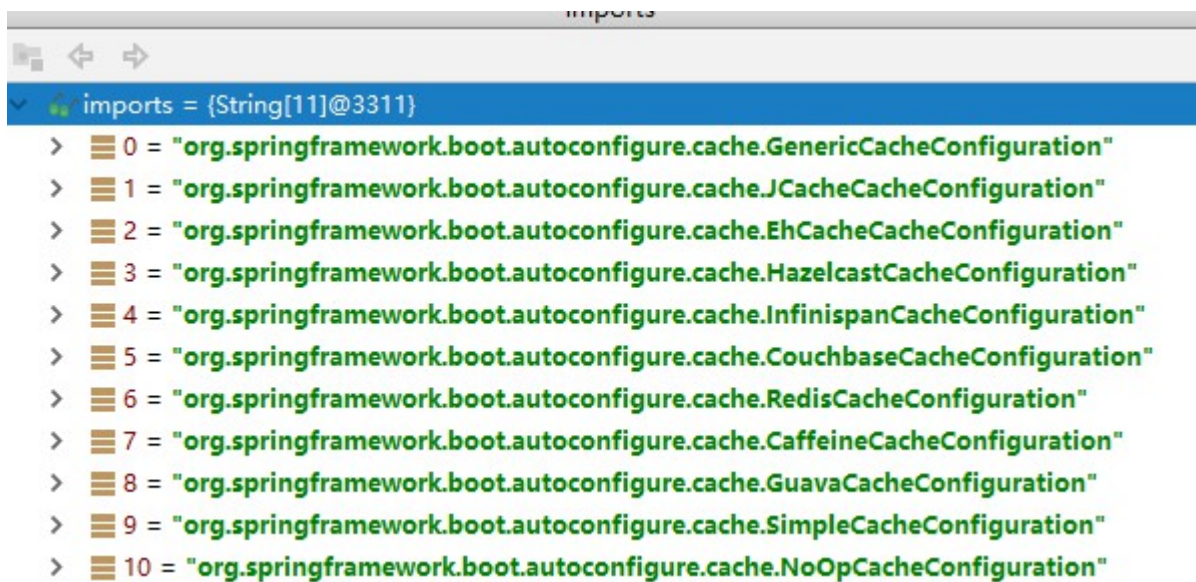
第一次访问会查询数据库，继续访问缓存中取值

3、缓存原理

1、原理：

1、CacheAutoConfiguration

2、导入缓存组件



3、查看哪个缓存配置生效

SimpleCacheConfiguration生效

4、给容器注册一个CacheManager:ConcurrentMapCacheManager

5、可以获取和创建ConcurrentMapCache,作用是将数据保存在ConcurrentMap中

2、运行流程

1、方法运行之前，先查Cache(缓存组件)，按照cacheName的指定名字获取；

(CacheManager先获取相应的缓存)，第一次获取缓存如果没有cache组件会自己创建

2、去Cache中查找缓存的内容，使用一个key，默认就是方法的参数；

key是按照某种策略生成的，默认是使用keyGenerator生成的，默认使用SimpleKeyGenerator生成key

没有参数 key=new SimpleKey()

如果有一个参数 key=参数值

如果多个参数 key=new SimpleKey(params);

3、没有查到缓存就调用目标方法

4、将目标方法返回的结果，放回缓存中

方法执行之前，@Cacheable先来检查缓存中是否有数据，按照参数的值作为key去查询缓存，如果没有，就运行方法，存入缓存，如果有数据，就取出map的值。

3、自定义缓冲KeyGenerator

```
/**
 * @Author: cuzz
 * @Date: 2018/9/26 18:39
 * @Description:
```

```

    */
@Configuration
public class MyCacheConfig {

    @Bean("myKeyGenerator")
    public KeyGenerator keyGenerator() {
        return new KeyGenerator() {
            @Override
            public Object generate(Object o, Method method, Object... objects) {
                return method.getName() + "[" + Arrays.asList(objects) + "]";
            }
        };
    }
}

```

指定keyGenerator

```

@Service
public class EmployeeService {

    @Autowired
    EmployeeMapper employeeMapper;

    @Cacheable(cacheNames = "emp", keyGenerator = "myKeyGenerator")
    public Employee getEmployee(Integer id) {
        System.out.println("----> 查询" + id + "号员工");
        return employeeMapper.getEmployeeById(id);
    }
}

```

4、Cache的注解

1、@Cacheput

修改数据库的某个数据，同时更新缓存

运行时机

先运行方法，再将目标结果缓存起来

1、编写更新方法

```

@CachePut(value = {"emp"})
public Employee updateEmployee(Employee employee) {
    System.out.println("---->updateEmployee"+employee);
    employeeMapper.updateEmp(employee);
    return employee;
}

```


2、编写Controller方法

```
@GetMapping("/emp")
public Employee update(Employee employee) {
    return employeeService.updateEmployee(employee);
}
```

测试

测试步骤

- 1、先查询1号员工
- 2、更新1号员工数据
- 3、查询1号员工

可能并没有更新,

是因为查询和更新的key不同

cacheable的key是不能使用result的参数

所以要使@CachePut使用@Cacheable使用相同的key

```
@CachePut(value = {"emp"}, key = "#result.id")
public Employee updateEmployee(Employee employee) {
    System.out.println("---->updateEmployee"+employee);
    employeeMapper.updateEmp(employee);
    return employee;
}
```

前面用的是Id 后面用的是Employee

效果:

- 第一次查询: 查询mysql
- 第二次更新: 更新mysql
- 第三次查询: 调用内存

2、CacheEvict

清除缓存

编写测试方法

service

```
@CacheEvict(value = "emp",key = "#id")
public void deleteEmployee(Integer id){
    System.out.println("---->删除的employee的id是: "+id);
}
```

controller

```
@GetMapping("/delete")
public String delete(Integer id) {
    employeeService.deleteEmployee(id);
    return "success";
}
```

allEntries = true, 代表不论清除那个key, 都重新刷新缓存

beforeInvocation=true 方法执行前, 清空缓存, 默认是false,如果程序异常, 就不会清除缓存

3、Caching定义组合复杂注解

组合

- Cacheable
- CachePut
- CacheEvict

CacheConfig抽取缓存的公共配置

```
@Caching(
    cacheable = {
        @Cacheable(value = "emp",key = "#lastName")
    },
    put = {
        @CachePut(value = "emp",key = "#result.id"),
        @CachePut(value = "emp",key = "#result.gender")
    }
)
public Employee getEmployeeByLastName(String lastName) {
    return employeeMapper.getEmpByLastName(lastName);
}
```

如果查完lastName, 再查的id是刚才的值, 就会直接从缓存中获取数据

4、CacheConfig抽取缓存的公共配置

```
@CacheConfig(cacheNames = "emp")
@Service
public class EmployeeService {
```

然后下面的value=emp就不用写了

5、Redis

默认的缓存是在内存中定义HashMap, 生产中使用Redis的缓存中间件

Redis 是一个开源 (BSD许可) 的, 内存中的数据结构存储系统, 它可以用作数据库、缓存和消息中间件

1、安装Docker

安装redis在docker上

```
#拉取redis镜像
docker pull registry.docker-cn.com/library/redis
#启动redis[e1a73233e3be]docker images的id
docker run -d -p 6379:6379 --name redis01 e1a73233e3be
```

2、Redis的Template

Redis的常用五大数据类型

String【字符串】、List【列表】、Set【集合】、Hash【散列】、ZSet【有序集合】

分为两种一种是StringRedisTemplate，另一种是RedisTemplate

根据不同的数据类型，大致的操作也分为这5种，以StringRedisTemplate为例

```
stringRedisTemplate.opsForValue() --String
stringRedisTemplate.opsForList() --List
stringRedisTemplate.opsForSet() --Set
stringRedisTemplate.opsForHash() --Hash
stringRedisTemplate.opsForZset() --Zset
```

1、导入依赖

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

2、修改配置文件

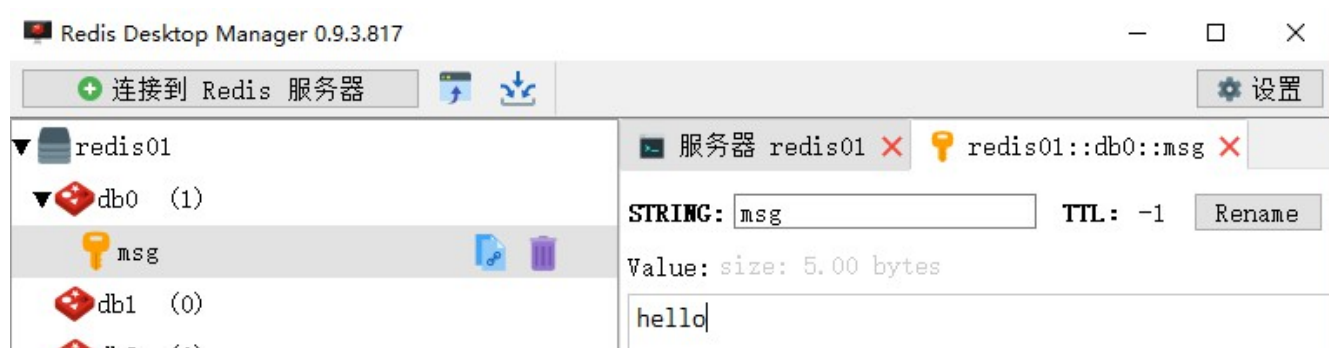
```
spring.redis.host=192.168.179.131
```

3、添加测试类

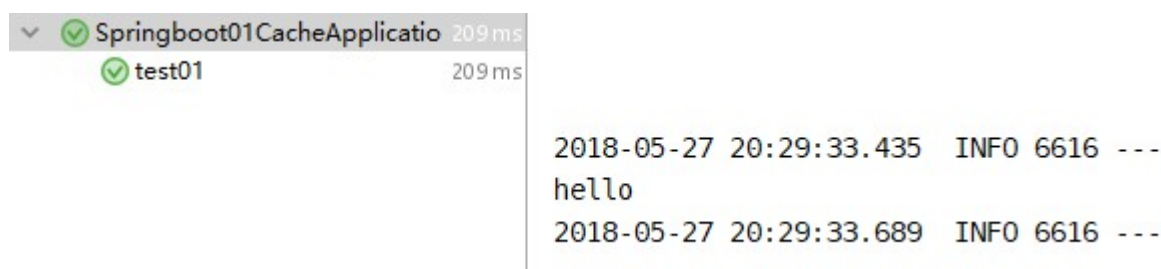
```
@Autowired
StringRedisTemplate stringRedisTemplate;//操作字符串【常用】

@Autowired
RedisTemplate redisTemplate;//操作k-v都是对象
@Test
public void test01(){
    // stringRedisTemplate.opsForValue().append("msg", "hello");
    String msg = stringRedisTemplate.opsForValue().get("msg");
    System.out.println(msg);
}
```

写入数据



读取数据



3、测试保存对象

对象需要序列化

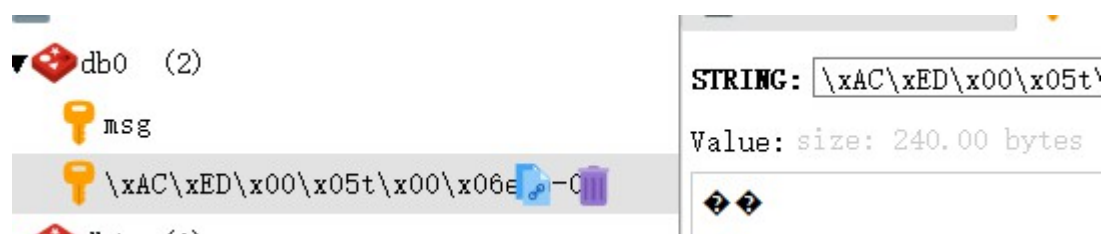
1、序列化bean对象

```
public class Employee implements Serializable {
```

2、将对象存储到Redis

```
@Test
public void test02(){
    Employee emp = employeeMapper.getEmpById(2);
    redisTemplate.opsForValue().set("emp-01", emp);
}
```

3、效果演示



4、以json方式传输对象

1、新建一个Redis的配置类MyRedisConfig,

```

/**
 * @Author: cuzz
 * @Date: 2018/9/26 23:50
 * @Description:
 */
@Configuration
public class MyRedisConfig {
    @Bean
    public RedisTemplate<Object, Employee> empRedisTemplate(
        RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Employee> jsonRedisSerializer = new
        Jackson2JsonRedisSerializer<Employee>(Employee.class);
        template.setDefaultSerializer(jsonRedisSerializer);
        return template;
    }
}

```

2、编写测试类

```

@Autowired
RedisTemplate<Object, Employee> empRedisTemplate;
@Test
public void test02(){
    Employee emp = employeeMapper.getEmpById(2);
    empRedisTemplate.opsForValue().set("emp-01", emp);
}

```

3、测试效果

The screenshot shows the Redis Desktop Manager interface. On the left, the tree view shows the Redis instance 'redis01' with database 'db0' selected. A key 'emp-01' is highlighted. On the right, the details pane shows the key type as 'STRING' and the value as a JSON object: `{"id":2,"lastName":"wanghuhu","gender":"1","email":null,"dId":null}`. The value size is 67.00 bytes.

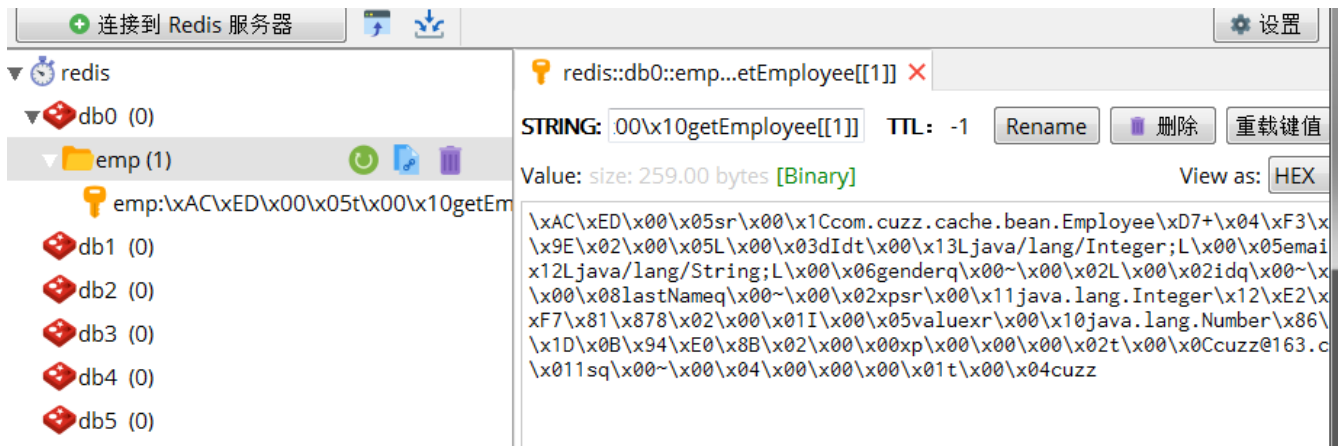
5、整合redis作为缓存

使用缓存时，默认使用的是ConcurrentMapCache，将数据保存在ConcurrentMap中，开发中使用的是缓存中间件，redis、memcached、ehcache等

starter启动时，有顺序，redis优先级比ConcurrentMapCache更高，CacheManager变为RedisCacheManager，所以使用的是redis缓存

传入的是RedisTemplate<Object, Object>

默认使用的是jdk的序列化保存



我们可以自定义CacheManager

编写一个配置类

```
/**
 * @Author: cuzz
 * @Date: 2018/9/26 23:50
 * @Description:
 */
@Configuration
public class MyRedisConfig {
    @Bean
    public RedisTemplate<Object, Employee> empRedisTemplate(
        RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Employee> jsonRedisSerializer = new
        Jackson2JsonRedisSerializer<Employee>(Employee.class);
        template.setDefaultSerializer(jsonRedisSerializer);
        return template;
    }

    @Bean
    public RedisCacheManager employeeCacheManager(RedisTemplate<Object, Employee>
empRedisTemplate) {
        RedisCacheManager cacheManager = new RedisCacheManager(empRedisTemplate);
        // 使用前缀，默认将CacheName作为前缀
        cacheManager.setUsePrefix(true);
        return cacheManager;
    }
}
```

可以看出变为json格式了

The screenshot shows the Redis Desktop Manager interface. On the left, a tree view shows the Redis instance structure: 'redis' (root) -> 'db0 (0)' -> 'emp (1)' -> 'emp:1'. The 'emp:1' key is selected, and its details are shown on the right. The key type is 'STRING', and the value is a JSON object:

```
{
  "id": 1,
  "lastName": "cuzz",
  "gender": "1",
  "email": "cuzz@163.com",
  "dId": 2
}
```

 The value size is 70.00 bytes.

empRedisTemplate只能用来反序列化Employee，而不能反序列化Department对象，因此要创建不同的xxxRedisTemplate

分别创建Manager

```
/**
 * @Author: cuzz
 * @Date: 2018/9/26 23:50
 * @Description:
 */
@Configuration
public class MyRedisConfig {
    // Employee
    @Bean
    public RedisTemplate<Object, Employee> empRedisTemplate(
        RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Employee> jsonRedisSerializer = new
        Jackson2JsonRedisSerializer<Employee>(Employee.class);
        template.setDefaultSerializer(jsonRedisSerializer);
        return template;
    }
    // Employee
    @Primary // 有多个cacheManager需要制定一个默认的一般将jdk的cacheManager作为默认
    @Bean
    public RedisCacheManager employeeCacheManager(RedisTemplate<Object, Employee>
empRedisTemplate) {
        RedisCacheManager cacheManager = new RedisCacheManager(empRedisTemplate);
        cacheManager.setUsePrefix(true);
        return cacheManager;
    }
}
```

```

    }

    // Department
    @Bean
    public RedisTemplate<Object, Department> deptRedisTemplate(
        RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<Object, Department> template = new RedisTemplate<Object, Department>
();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Department> jsonRedisSerializer = new
Jackson2JsonRedisSerializer<Department>(Department.class);
        template.setDefaultSerializer(jsonRedisSerializer);
        return template;
    }

    // Department
    @Bean
    public RedisCacheManager departmentCacheManager(RedisTemplate<Object, Department>
deptRedisTemplate) {
        RedisCacheManager cacheManager = new RedisCacheManager(deptRedisTemplate);
        cacheManager.setUsePrefix(true);
        return cacheManager;
    }
}

```

指定CacheManager

```

/**
 * @Author: cuzz
 * @Date: 2018/9/26 16:08
 * @Description:
 */
@CacheConfig(cacheNames = "emp", cacheManager = "employeeCacheManager")
@Service
public class EmployeeService {}

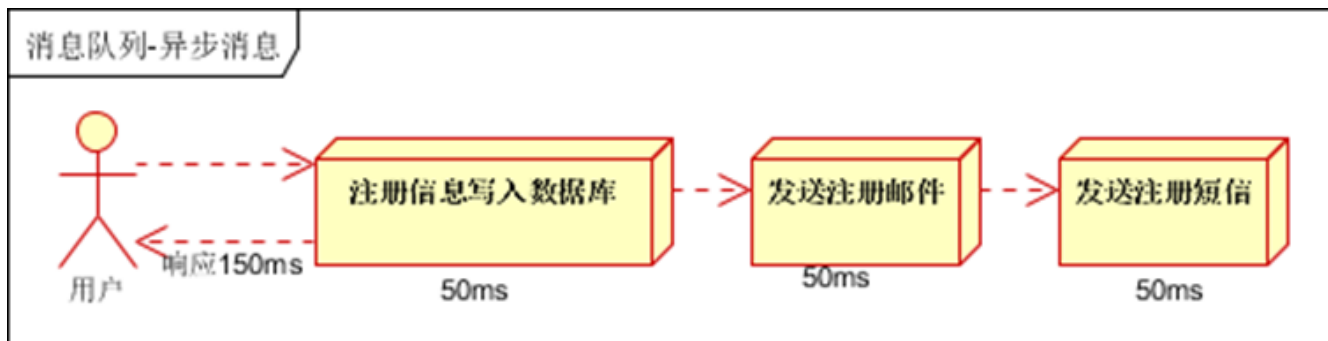
```

十、SpringBoot的消息中间件

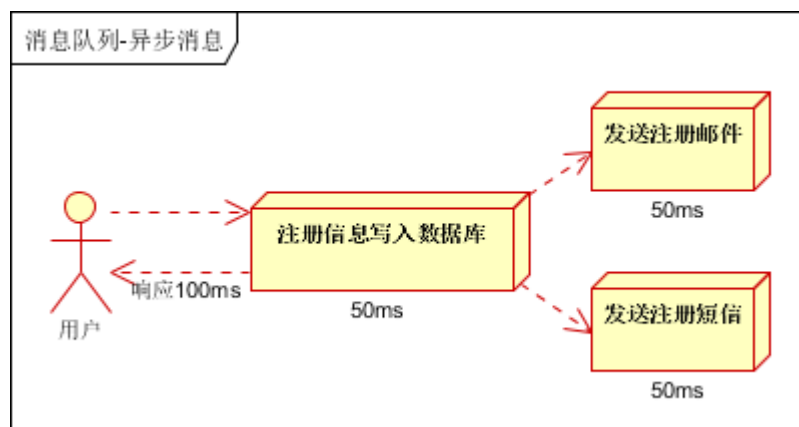
1、JMS&AMQP简介

1、异步处理

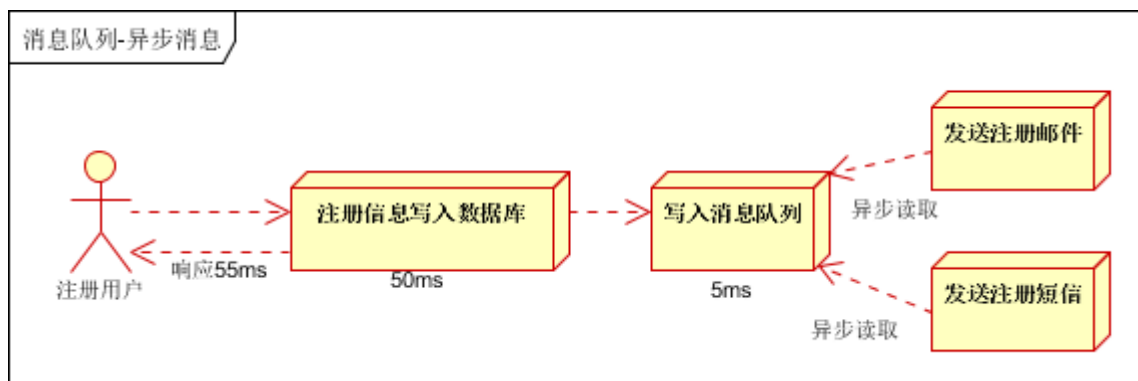
同步机制



并发机制

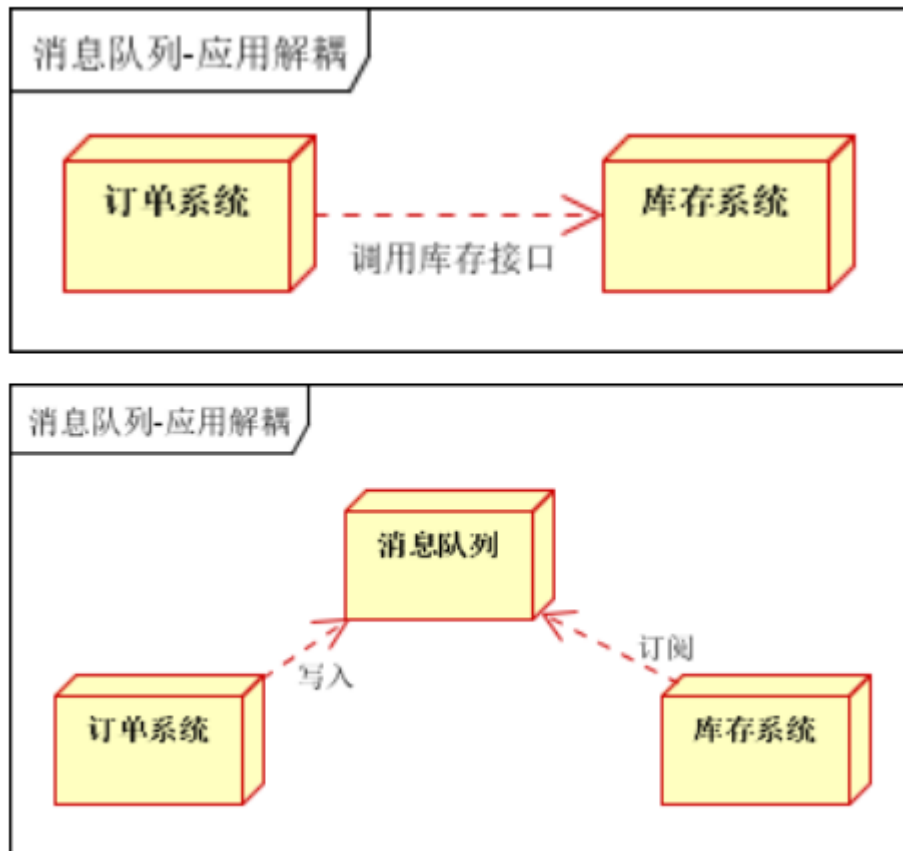


消息队列机制



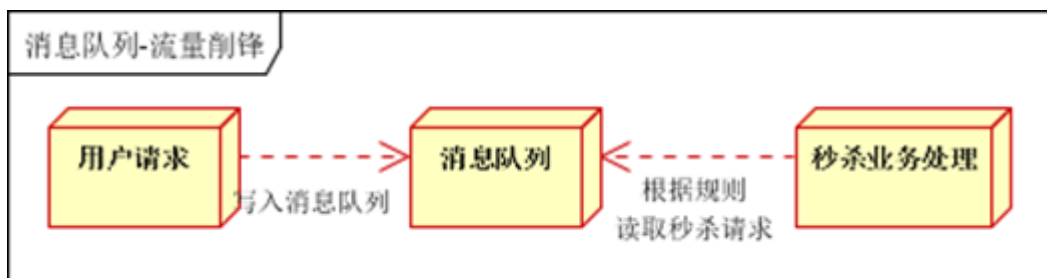
2、应用解耦

使用中间件，将两个服务解耦，一个写入，一个订阅



3、流量削锋

例如消息队列的FIFO，限定元素的长度，防止出现多次请求导致的误操作



概述

1、大多数应用，可以通过消息服务中间件来提升系统的异步通信、拓展解耦能力

2、消息服务中的两个重要概念：

消息代理（message broker）和目的地（destination），当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定的目的地。

3、消息队列主要的两种形式的目的地

1)、队列（queue）：点对点消息通信【point-to-point】，取出一个没一个，一个发布，多个消费

2)、主题（topic）：发布（publish）/订阅（subscribe）消息通信，多人【订阅者】可以同时接到消息

4、JMS(Java Message Service) Java消息服务：

- 基于JVM消息规范的代理。ActiveMQ/HornetMQ是JMS的实现

5、AMQP(Advanced Message Queuing Protocol)

- 高级消息队列协议，也是一个消息代理的规范，兼容JMS
- RabbitMQ是AMQP的实现

	JMS	AMQP
定义	Java API	网络线级协议
跨平台	否	是
跨语言	否	是
Model	(1)、Peer-2-Peer (2)、Pub/Sub	(1)、direct exchange (2)、fanout exchange (3)、topic change (4)、headers exchange (5)、system exchange 后四种都是pub/sub ,差别路由机制做了更详细的划分
支持消息类型	TextMessage MapMessage ByteMessage StreamMessage ObjectMessage Message	byte[]通常需要序列化

6、SpringBoot的支持

spring-jms提供了对JMS的支持

spring-rabbit提供了对AMQP的支持

需要创建ConnectionFactory的实现来连接消息代理

提供JmsTemplate,RabbitTemplate来发送消息

@JmsListener(JMS).@RabbitListener(AMQP)注解在方法上的监听消息代理发布的消息

@EnableJms,@EnableRabbit开启支持

7、SpringBoot的自动配置

- JmsAutoConfiguration
- RabbitAutoConfiguration

2、RabbitMQ简介

AMQP的实现

1、核心概念

Message:消息头和消息体组成，消息体是不透明的，而消息头上则是由一系列的可选属性组成，属性：路由键【routing-key】,优先级【priority】,指出消息可能需要持久性存储【delivery-mode】

Publisher:消息的生产者，也是一个向交换器发布消息的客户端应用程序

Exchange:交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列

Exchange的4中类型：direct【默认】点对点，fanout,topic和headers,发布订阅，不同类型的Exchange转发消息的策略有所区别

Queue:消息队列，用来保存消息直到发送给消费者，它是消息的容器，也是消息的终点，一个消息可投入一个或多个队列，消息一直在队列里面，等待消费者连接到这个队列将数据取走。

Binding:绑定，队列和交换机之间的关联，多对多关系

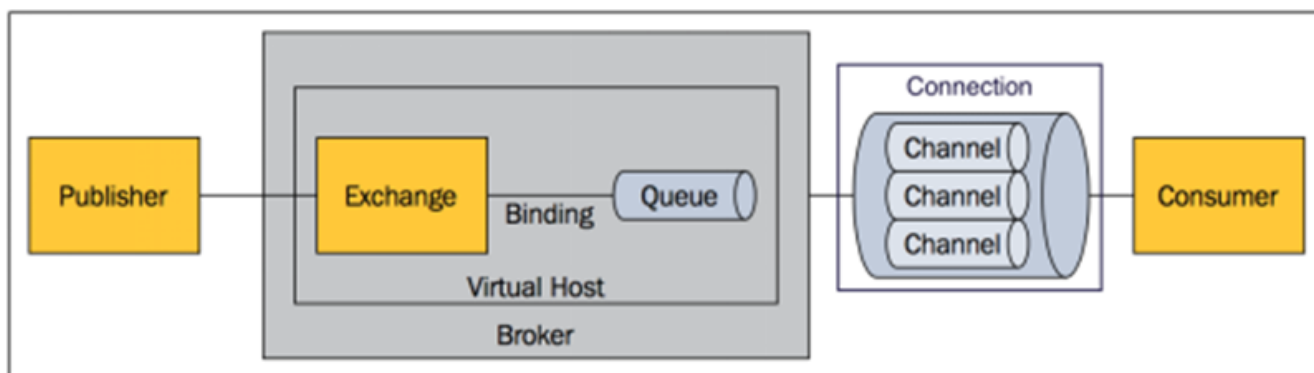
Connection:网络连接，例如TCP连接

Channel:信道，多路复用连接中的一条独立的双向数据流通道，信道是建立在真的TCP链接之中的虚拟连接AMQP命令都是通过信道发送出去的。不管是发布消息，订阅队列还是接受消息，都是信道，减少TCP的开销，复用一条TCP连接。

Consumer:消息的消费者，表示一个从消息队列中取得消息的客户端的应用程序

VirtualHost:虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 /。

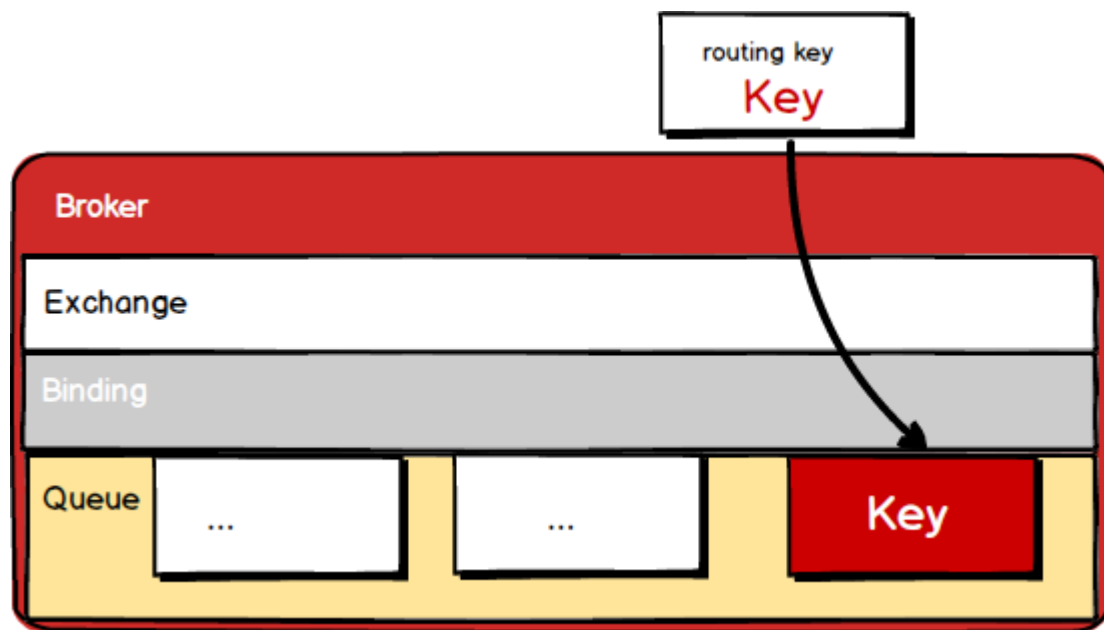
Broker:表示消息队列 服务实体



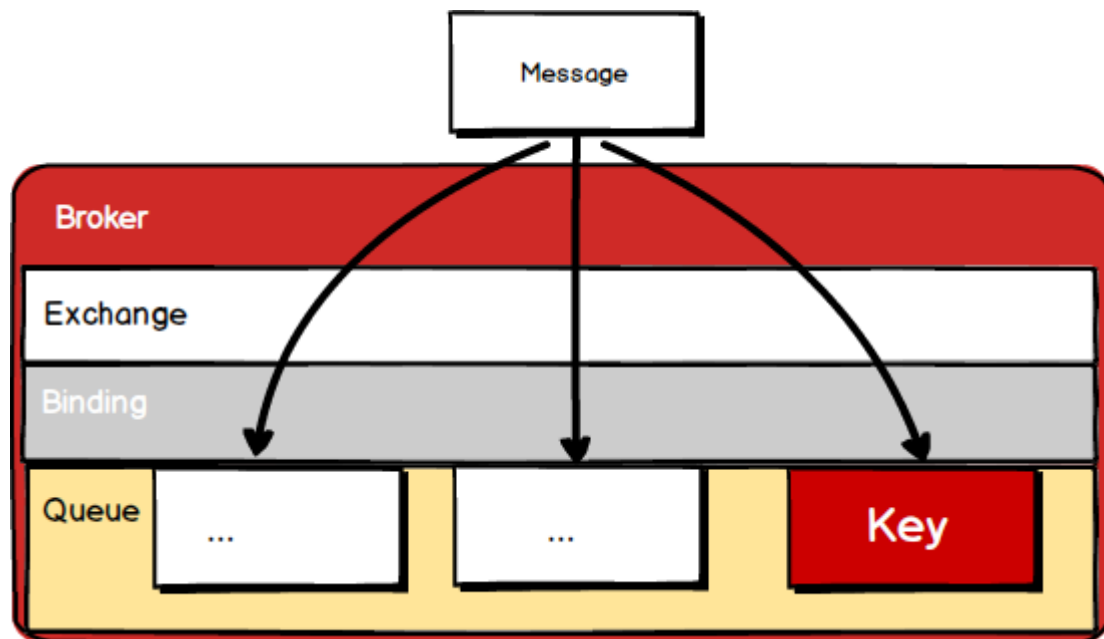
2、RabbitMQ的运行机制

Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers。headers 匹配 AMQP 消息的 header 而不是路由键，headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

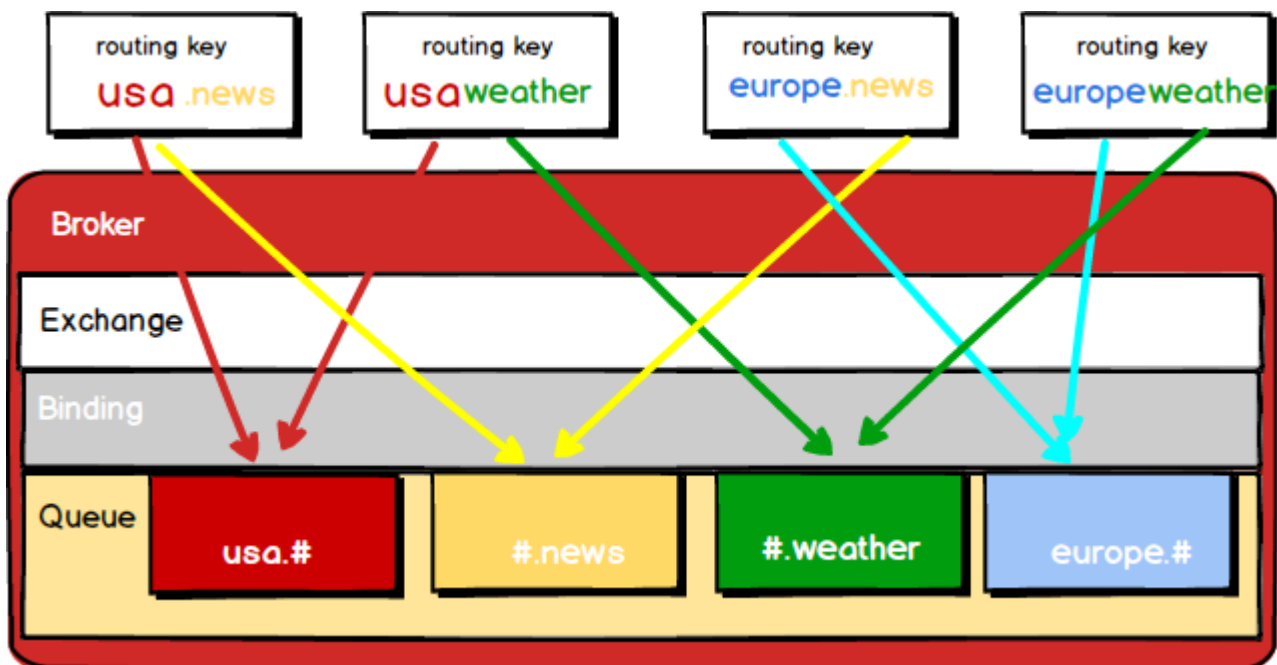
direct: 根据路由键直接匹配，一对一



fanout: 不经过路由键，直接发送到每一个队列



topic: 类似模糊匹配的根据路由键，来分配绑定的队列



3、RabbitMQ安装测试

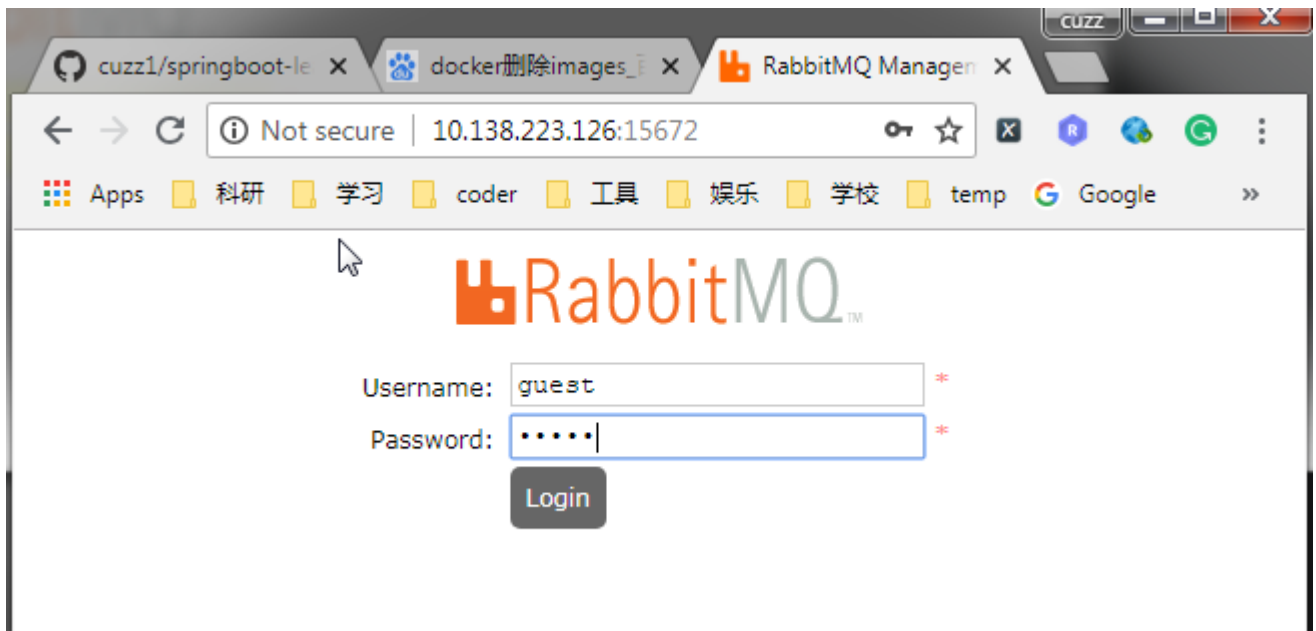
1、打开虚拟机，在docker中安装RabbitMQ

```
#1.安装rabbitmq, 使用镜像加速
docker pull registry.docker-cn.com/library/rabbitmq:3-management
[root@node1 ~]# docker images
REPOSITORY                                TAG                IMAGE ID
CREATED                                SIZE
registry.docker-cn.com/library/rabbitmq    3-management       e1a73233e3be      11
days ago                                149 MB
#2.运行rabbitmq
##### 端口: 5672 客户端和rabbitmq通信 15672: 管理界面的web页面

docker run -d -p 5672:5672 -p 15672:15672 --name myrabbitmq e1a73233e3be

#3.查看运行
docker ps
```

2、打开网页客户端并登陆，账号【guest】，密码【guest】，登陆



3、添加【direct】【faout】【topic】的绑定关系等

1)、添加Exchange,分别添加**exchange.direct**、**exchange.fanout**、**exchange.topic**

exchange.direct	direct	D	0.00/s	0.00/s
exchange.fanout	fanout	D	0.00/s	0.00/s
exchange.topic	topic	D		

▼ Add a new exchange

Name:

Type: → exchange的4中方式

Durability: → 持久化, 下次打开还是存储到这里, 不会清除

Auto delete: ?

Internal: ?

Arguments: = String ▼

Add Alternate exchange ?

2)、添加 Queues,分别添加**cuzz.news**、**cuzz**、**cuzz.emps**、**cuxx.news**

Queues

► All queues (4)

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
cuxx.news	D	idle	0	0	0			
cuzz	D	idle	0	0	0			
cuzz.emps	D	idle	0	0	0			
cuzz.news	D	idle	0	0	0			

▼ Add a new queue

Name: *

Durability: Durable ▼

Auto delete: ? No ▼

Arguments: = String ▼

Add Message TTL ? | Auto expire ? | Max length ? | Max length bytes ? |

3)、点击【exchange.direct】添加绑定规则

▼ Bindings

This exchange



To	Routing key	Arguments	
cuzz	cuzz		Unbind
cuzz.emps	cuzz.emps		Unbind
cuzz.news	cuzz.news		Unbind

Add binding from this exchange

To queue ▼ : *

Routing key:

Arguments: =

Bind

4)、点击【exchange.fanout】添加绑定规则

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
cuxx.news	cuxx.news		Unbind
cuzz	cuzz		Unbind
cuzz.emps	cuzz.emps		Unbind
cuzz.news	cuzz.news		Unbind

5)、点击【exchange.topic】添加绑定规则

▼ Bindings

This exchange

⇓

⏱

To	Routing key	Arguments	
cuxx.news	*.news		Unbind
cuzz	cuzz.#		Unbind
cuzz.emps	cuzz.#		Unbind
cuzz.news	*.news		Unbind
cuzz.news	cuzz.#		Unbind

/*: 代表匹配1个单词

/#: 代表匹配0个或者多个单词

4、发布信息测试

【direct】发布命令，点击 Publish message

▼ Publish message

Routing key:

Delivery mode: 1 - Non-persistent ▼

Headers: ?


=

String ▼

Properties: ?

=

Payload:



Publish message

查看队列的数量

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
cuxx.news	D	idle	0	0	0				
cuzz	D	idle	1	0	1	0.00/s			
cuzz.emps	D	idle	0	0	0				
cuzz.news	D	idle	0	0	0				

点击查看发送的信息，点击Get Message

Overview

Connections

Channels

Exchanges

Queues

Admin

Ack Mode:

Nack message requeue true

Encoding:

Auto string / base64

 ?

Messages:

1

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange

Routing Key

Redelivered

Properties

Payload

30 bytes

Encoding: string

exchange.direct

cuzz

0

delivery_mode: 1

headers:

direct.exchange.msg.helloworld

【fanout】的发布消息

▼ Publish message

Routing key:

cuzz.news

Delivery mode:

1 - Non-persistent

Headers: ?

=

String

Properties: ?

=

Payload:

fanout.msg.hello

队列信息，每个队列都添加了一条

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
cuzz.news	D	idle	1	0	1	0.00/s		
cuzz	D	idle	2	0	2	0.20/s	0.00/s	0.00/s
cuzz.emps	D	idle	1	0	1	0.00/s		
cuzz.news	D	idle	1	0	1	0.00/s		

【topic】发布信息测试

▼ Publish message

Routing key:

Delivery mode:

Headers: ? =

Properties: ? =

Payload:

队列的值

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
cuxx.news	D	idle	2	0	2	0.00/s			
cuzz	D	idle	3	0	3	0.00/s	0.00/s	0.00/s	
cuzz.emps	D	idle	2	0	2	0.00/s			
cuzz.news	D	idle	2	0	2	0.00/s			

cuzz.news 可以匹配 cuzz.# 也可以匹配 *.news 所以都能匹配到

信息查看

4、创建工程整合

1、RabbitAutoConfiguration 2、自动配置了连接工厂 ConnectionFactory 3、RabbitProperties封装了 RabbitMQ 4、RabbitTemplate:给RabbitMQ发送和接受消息的 5、AmqpAdmin：RabbitMQ的系统管理功能组件

1、RabbitTemplate

- 1、新建SpringBoot工程，SpringBoot1.5+Inteegration/RabbitMQ+Web
- 2、RabbitAutoConfiguration文件
- 3、编写配置文件application.yml

```
spring:
  rabbitmq:
    host: 10.138.223.126
    port: 5672
    username: guest
    password: guest
```

- 4、编写测试类,将HashMap写入Queue

```
@Test
public void contextLoads() {
    // Message需要自己构建一个；定义消息体内容和消息头
    // rabbitTemplate.send(exchange, routingKey, message);
    // Object 默认当成消息体，只需要传入要发送的对象，自动化序列发送给rabbitmq；
    Map<String,Object> map = new HashMap<>();
    map.put("msg", "这是第一个信息");
    map.put("data", Arrays.asList("HelloWorld", 123, true));
    //对象被默认序列以后发送出去
    rabbitTemplate.convertAndSend("exchange.direct","cuzz.news",map);
}
```

5、查看网页的信息，默认使用java序列的方式

```
Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange      exchange.direct
Routing Key    cuzz.news
Redelivered    0
Properties
  priority: 0
  delivery_mode: 2
  headers:
    content_type: application/x-java-serialized-object

Payload
  365 bytes
  Encoding: base64
  rO0ABXNyABFqYXZlbnV0aWwusGFzaE1hcAUH2sHDFmDRAwACRgAKbG9hZEZhY3RvcckACX RocmVzaG9sZHHwP0AAAAAAAAAx3CAAAABAAAAACdAAdbXNndAAV
  6L+Z5piv56ys5LiA5Liq5L+h5oGvdAAEZGF0YXNyABpqYXZlbnV0aWwusQXJyYXlzlEFycmFSTGlzdNmkPL7NiAbSAGABWwABYXQAE1tMamF2YS9sYW5nL09i
  amVjdDt4dHVyABdbTGphdmEuaW8uU2VyaWFsaXphYmxlO67QCaxT1+1JAgAAeHAAAAADdAAKSgVsbG9Xb3JsZHNyABFqYXZlbnV0aWwusSW50ZWdlchLioKT3
  gYc4AgABSQAfmdFsdWV4cgAQamF2YS5sYW5nLk51bWJlcoasIR0LIOCLagAAeHAAAAAB7c3IAEWphdmEubGFuZy5Cb29sZWZuzSBygNWc+u4CAAFaAAV2YWx1
  ZXhwAXg=
```

6、取出队列的值

取出队列中数据就没了

```
@Test
public void receiveAndConvert(){
    Object o = rabbitTemplate.receiveAndConvert("cuzz.news");
    System.out.println(o.getClass());
    System.out.println(o);
}
```

结果

```
class java.util.HashMap
{msg=这是第一个信息, data=[Hello world, 123, true]}
```

7、使用Json方式传递，并传入对象Book

1)、MyAMQPConfig, 自定义一个MessageConverter返回Jackson2JsonMessageConverter

```

/**
 * @Author: cuzz
 * @Date: 2018/9/27 14:16
 * @Description:
 */
@Configuration
public class MyAMQPConfig {

    @Bean
    public MessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }
}

```

发现已经转化为json了

Message 1

The server reported 0 messages remaining.

Exchange	exchange.direct
Routing Key	cuzz.news
Redelivered	0
Properties	priority: 0 delivery_mode: 2 headers: __ContentTypeId__: java.lang.Object __KeyTypeId__: java.lang.Object __TypeId__: java.util.HashMap content_encoding: UTF-8 content_type: application/json
Payload 62 bytes Encoding: string	<div> {"msg":"这是第一个信息","data":["HelloWorld",123,true]} </div>

2)、编写Book实体类

```

/**
 * @Author: cuzz
 * @Date: 2018/9/27 14:22
 * @Description:
 */
@Data
public class Book {
    private String bookName;
    private String author;

    public Book(){
    }

    public Book(String bookName, String author) {

```

```

        this.bookName = bookName;
        this.author = author;
    }
}

```

3)、测试类

```

@Test
public void test() {
    // 对象被默认序列以后发送出去
    rabbitTemplate.convertAndSend("exchange.direct","cuzz.news", new Book("Effect java",
"Joshua Bloch"));
}

```

4)、查看cuzz.news

5)、取出数据

```

@Test
public void reciverAndConvert(){
    Object o = rabbitTemplate.receiveAndConvert("cuzz.news");
    System.out.println(o.getClass());
    System.out.println(o);
}

```

6)、结果演示

```

class com.cuzz.amqp.bean.Book
Book(bookName=Effect java, author=Joshua Bloch)

```

2、开启基于注解的方式

1、新建一个BookService

```

@Service
public class BookService {
    @RabbitListener(queues = "cuzz.news")
    public void receive(Book book){
        System.out.println(book);
    }

    @RabbitListener(queues = "cuzz")
    public void receive02(Message message){
        System.out.println(message.getBody());
        System.out.println(message.getMessageProperties());
    }
}

```

2、主程序开启RabbitMQ的注解

```

@EnableRabbit // 开启基于注解的rabbitmq
@SpringBootApplication
public class Springboot10AmqpApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot10AmqpApplication.class, args);
    }
}

```

3、AmqpAdmin

创建和删除 Exchange、Queue、Bind

1)、创建Exchange

```

@Test
public void createExchange(){
    amqpAdmin.declareExchange(new DirectExchange("amqpadmin.direct"));
    System.out.println("Create Finish");
}

```

效果演示

amq.topic	topic	D		
amqpadmin.direct	direct	D		
exchange.direct	direct	D	0.00/s	0.00/s
exchange.fanout	fanout	D	0.00/s	0.00/s
exchange.topic	topic	D	0.00/s	0.00/s

2)、创建Queue

```

@Test
public void createQueue(){
    amqpAdmin.declareQueue(new Queue("amqpadmin.queue",true));
    System.out.println("Create Queue Finish");
}

```

amqpadmin.queue	D	idle	0	0	0			
lxy.news	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
wdjr	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
wdjr.emps	D	idle	1	0	1	0.00/s	0.00/s	0.00/s
wdjr.news	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

3)、创建Bind规则


```
@Test
public void createBind(){
    amqpAdmin.declareBinding(new Binding("amqpadmin.queue",Binding.DestinationType.QUEUE ,
"amqpadmin.direct", "amqp.haha", null));
}
```

To	Routing key	Arguments	
amqpadmin.queue	amqp.haha		Unbind

删除类似

```
@Test
public void deleteExchange(){
    amqpAdmin.deleteExchange("amqpadmin.direct");
    System.out.println("delete Finish");
}
```

十一、SpringBoot的检索

1、ElasticSearch简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web 接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于[云计算](#)中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

2、ElasticSearch的安装

1、安装java最新版本

- 下载linux的.tar.gz
- 解压到指定目录
- 配置环境变量

2、安装Docker(非必须这是是在Docker中安装)

```
1、查看centos版本
# uname -r
3.10.0-693.el7.x86_64
要求：大于3.10
如果小于的话升级*（选做）
# yum update
2、安装docker
# yum install docker
3、启动docker
# systemctl start docker
# docker -v
4、开机启动docker
# systemctl enable docker
```

```
5、停止docker
# systemctl stop docker
```

3、安装ElasticSearch的镜像

```
docker pull registry.docker-cn.com/library/elasticsearch
```

4、运行ElasticSearch

-e ES_JAVA_OPTS="-Xms256m -Xmx256m" 表示占用的最大内存为256m，默认是2G

```
[root@node1 ~]# docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300
--name ES01 5acf0e8da90b
```

5、测试是否启动成功

访问9200端口: <http://10.138.223.126:9200/> 查看是否返回json数据

```
{
  "name" : "DNF7ndJ",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "t-JMNI1LRgGf62ctJqiGYQ",
  "version" : {
    "number" : "5.6.12",
    "build_hash" : "cfe3d9f",
    "build_date" : "2018-09-10T20:12:43.732Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}
```

3、Elastic的快速入门

最好的工具就是[官方文档](#)，以下操作都在文档中进行操作。

1、基础概念

面向文档，JSON作为序列化格式，ElasticSearch的基本概念

索引 (名词)：

如前所述，一个 *索引* 类似于传统关系数据库中的一个 *数据库*，是一个存储关系型文档的地方。*索引*/(*index*) 的复数词为 *indices* 或 *indexes*。

索引 (动词)：

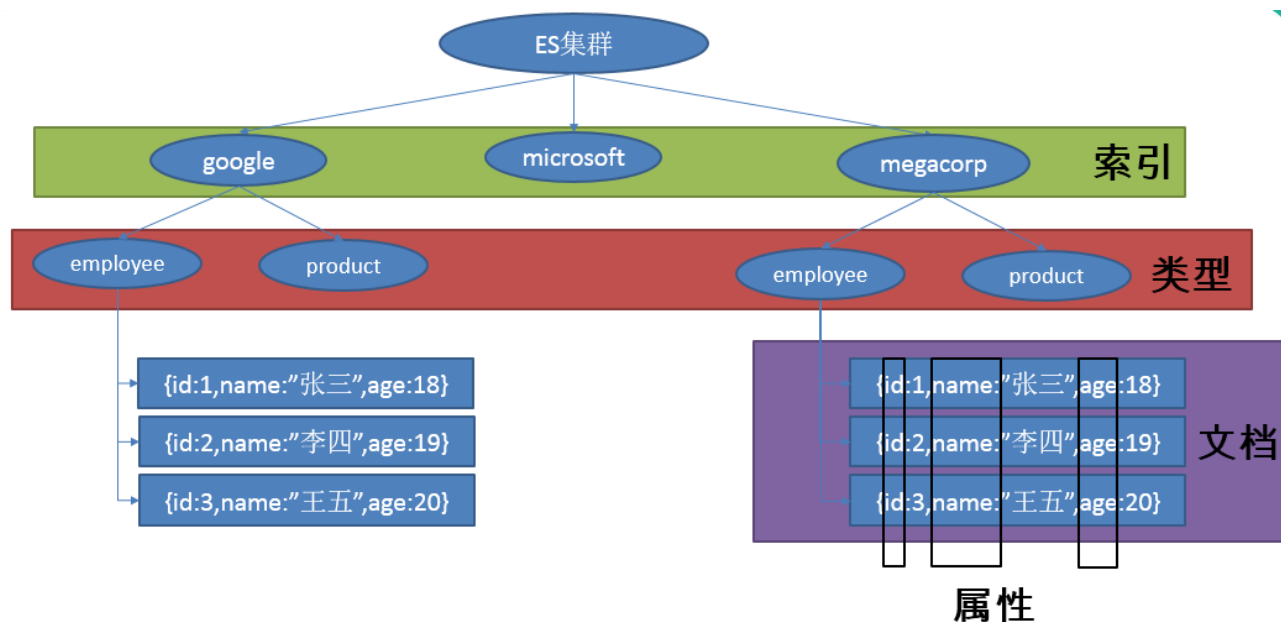
*索引*一个文档就是存储一个文档到一个 *索引* (名词) 中以便它可以被检索和查询到。这非常类似于 SQL 语句中的 `INSERT` 关键词，除了文档已存在时新文档会替换旧文档情况之外。

类型：相当于数据库中的表

文档：相当于数据库中的行，即每条数据都叫一个文档

属性：相当于数据库中的列，即文档的属性

结构图：



2、测试

下载[POSTMAN](#)，并使用POSTMAN测试

1、插入数据

具体信息查看[官方示例](#)

重点：PUT请求+请求体

```
PUT /megacorp/employee/1
{
  "first_name" : "John",
  "last_name"  : "Smith",
  "age"       : 25,
  "about"     : "I love to go rock climbing",
  "interests": [ "sports", "music" ]
}
```

http://10.138.223.126:9200/megacorp/employee/1

PUT

UI

form-data

x-www-form-urlencoded

raw

Text

```
1 {
2   "first_name" : "John",
3   "last_name" : "Smith",
4   "age" : 25,
5   "about" : "I love to go rock climbing",
6   "interests": [ "sports", "music" ]
7 }
```

Send

Preview

Add to collection

Body

Cookies (1)

Headers (5)

STATUS

201 Created

TIME

1733 ms

Pretty

Raw

Preview



JSON

XML

```
1 {
2   "_index": "megacorp",
3   "_type": "employee",
4   "_id": "1",
5   "_version": 1,
6   "result": "created",
7   "_shards": {
8     "total": 2,
9     "successful": 1,
10    "failed": 0
11  },
12   "created": true
13 }
```

2、检索文档

[官方示例](#)

重点：GET请求+URI+index+type+ID

GET /megacorp/employee/1

http://10.138.223.126:9200/megacorp/employee/1 GET

Send Preview Add to collection

Body Cookies (1) Headers (3) STATUS 200 OK TIME 49 ms

Pretty Raw Preview JSON XML

```
1 {
2   "_index": "megacorp",
3   "_type": "employee",
4   "_id": "1",
5   "version": 1,
6   "found": true,
7   "_source": {
8     "first_name": "John",
9     "last_name": "Smith",
10    "age": 25,
11    "about": "I love to go rock climbing",
12    "interests": [
13      "sports",
14      "music"
15    ]
16  }
17 }
```

3、轻量检索

[官方文档](#)

重点：GET请求+index+type+_search+条件（非必须）

搜索所有雇员： `_search`

```
GET /megacorp/employee/_search
```

高亮搜索：URL参数

```
GET /megacorp/employee/_search?q=last_name:Smith
```

4、使用查询表达式

[官方文档](#)

重点：GET+URI+index+type+_search+请求体【match】

Query-string 搜索通过命令非常方便地进行临时性的即席搜索，但它有自身的局限性（参见 [轻量搜索](#)）。Elasticsearch 提供一个丰富灵活的查询语言叫做 *查询表达式*，它支持构建更加复杂和健壮的查询。

领域特定语言（DSL），指定了使用一个 JSON 请求。我们可以像这样重写之前的查询所有 Smith 的搜索：

```
GET /megacorp/employee/_search
{
  "query" : {
    "match" : {
      "last_name" : "Smith"
    }
  }
}
```

返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 *query-string* 参数，而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 `match` 查询（属于查询类型之一，后续将会了解）。

5、更加复杂的查询

[官方文档](#)

重点：GET+URI+index+type+_search + 请求体【match+filter】

现在尝试下更复杂的搜索。同样搜索姓氏为 Smith 的雇员，但这次我们只需要年龄大于 30 的。查询需要稍作调整，使用过滤器 *filter*，它支持高效地执行一个结构化查询。

```
GET /megacorp/employee/_search
{
  "query" : {
    "bool": {
      "must": {
        "match" : {
          "last_name" : "smith"
        }
      },
      "filter": {
        "range" : {
          "age" : { "gt" : 30 }
        }
      }
    }
  }
}
```

- | | |
|---|--|
| ❶ | 这部分与我们之前使用的 <code>match</code> 查询一样。 |
| ❷ | 这部分是一个 <code>range</code> 过滤器，它能找到年龄大于 30 的文档，其中 <code>gt</code> 表示大于(great than)。 |

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个 *过滤器* 用于执行一个范围查询，并复用之前的 `match` 查询。现在结果只返回了一个雇员，叫 Jane Smith，32 岁。

6、全文搜索

[官方文档](#)

重点：GET+index+type+_search+请求体【match】 ==》看相关性得分

截止目前的搜索相对都很简单：单个姓名，通过年龄过滤。现在尝试下稍微高级点儿的全文搜索——一项传统数据库确实很难搞定的任务。

搜索下所有喜欢攀岩（rock climbing）的雇员：

```
GET /megacorp/employee/_search
{
  "query" : {
    "match" : {
      "about" : "rock climbing"
    }
  }
}
```

显然我们依旧使用之前的 `match` 查询在 `about` 属性上搜索 “rock climbing”。得到两个匹配的文档：

```
{
  ...
  "hits": {
    "total":      2,
    "max_score":  0.16273327,
    "hits": [
      {
        ...
        "_score":      0.16273327,
        "_source": {
          "first_name": "John",
          "last_name":  "Smith",
          "age":        25,
          "about":      "I love to go rock climbing",
          "interests":  [ "sports", "music" ]
        }
      },
      {
        ...
        "_score":      0.016878016,
        "_source": {
          "first_name": "Jane",
          "last_name":  "Smith",
          "age":        32,
          "about":      "I like to collect rock albums",
          "interests":  [ "music" ]
        }
      }
    ]
  }
}
```

“_score”:相关性得分

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。第一个最高得分的结果很明显：John Smith 的 `about` 属性清楚地写着 “rock climbing”。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 `about` 属性里提到了“rock”。因为只有“rock”而没有“climbing”，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何在全文属性上搜索并返回相关性最强的结果。Elasticsearch 中的 *相关性* 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

7、短语搜索

[官方文档](#)

重点：GET+index+type+_search+请求体【match_phrase】

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者 *短语*。比如，我们想执行这样一个查询，仅匹配同时包含“rock”和“climbing”，并且二者以短语“rock climbing”的形式紧挨着的雇员记录。

为此对 `match` 查询稍作调整，使用一个叫做 `match_phrase` 的查询：

```
GET /megacorp/employee/_search
{
  "query" : {
    "match_phrase" : {
      "about" : "rock climbing"
    }
  }
}
```

返回的信息

```
{
  ...
  "hits": {
    "total": 1,
    "max_score": 0.23013961,
    "hits": [
      {
        ...
        "_score": 0.23013961,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      }
    ]
  }
}
```

8、高亮搜索

[官方地址](#)

重点：GET+index+type+_search+请求体【match_phrase+highlight】==>返回关键字加了em标签

许多应用都倾向于在每个搜索结果中 高亮 部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。

再次执行前面的查询，并增加一个新的 `highlight` 参数：

```
GET /megacorp/employee/_search
{
  "query" : {
    "match_phrase" : {
      "about" : "rock climbing"
    }
  },
  "highlight": {
    "fields" : {
      "about" : {}
    }
  }
}
```

当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 `highlight` 的部分。这个部分包含了 `about` 属性匹配的文本片段，并以 HTML 标签 `` 封装：

```
{
  ...
  "hits": {
    "total": 1,
    "max_score": 0.23013961,
    "hits": [
      {
        ...
        "_score": 0.23013961,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        },
        "highlight": {
          "about": [
            "I love to go <em>rock</em> <em>climbing</em>"
          ]
        }
      }
    ]
  }
}
```

9、分析

[官方文档](#)

重点：GET+index+type+_search+请求体【aggs-field】

aggs: 聚合

终于到了最后一个业务需求：支持管理者对雇员目录做分析。Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 `GROUP BY` 类似但更强大。

举个例子，挖掘出雇员中最受欢迎的兴趣爱好：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

会报错

Fielddata is disabled on text fields by default. Set fielddata=true on [inte

默认情况下，字段数据在文本字段上禁用。设置字段数据= TRUE

首先开启数据结构

```
PUT megacorp/_mapping/employee/
{
  "properties": {
    "interests": {
      "type": "text",
      "fielddata": true
    }
  }
}
```

然后在进行请求

```
{
  ...
  "hits": { ... },
  "aggregations": {
    "all_interests": {
      "buckets": [
        {
          "key": "music",
          "doc_count": 2
        },
        {
          "key": "forestry",
          "doc_count": 1
        },
        {
          "key": "sports",
          "doc_count": 1
        }
      ]
    }
  }
}
```

```

    }
  ]
}
}
}

```

可以看到，两位员工对音乐感兴趣，一位对林地感兴趣，一位对运动感兴趣。这些聚合并非预先统计，而是从匹配当前查询的文档中即时生成。

如果想知道叫 Smith 的雇员中最受欢迎的兴趣爱好，可以直接添加适当的查询来组合查询：

```

GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}

```

`all_interests` 聚合已经变为只包含匹配查询的文档：

```

...
"all_interests": {
  "buckets": [
    {
      "key": "music",
      "doc_count": 2
    },
    {
      "key": "sports",
      "doc_count": 1
    }
  ]
}

```

聚合还支持分级汇总。比如，查询特定兴趣爱好员工的平均年龄：

```
GET /megacorp/employee/_search
{
  "aggs" : {
    "all_interests" : {
      "terms" : { "field" : "interests" },
      "aggs" : {
        "avg_age" : {
          "avg" : { "field" : "age" }
        }
      }
    }
  }
}
```

输出基本是第一次聚合的加强版。依然有一个兴趣及数量的列表，只不过每个兴趣都有了一个附加的 `avg_age` 属性，代表有这个兴趣爱好所有员工的平均年龄。

即使现在不太理解这些语法也没有关系，依然很容易了解到复杂聚合及分组通过 Elasticsearch 特性实现得很完美。可提取的数据类型毫无限制。

4、SpringBoot+ElasticSearch

1、新建项目SpringBoot1.5+Web+Nosql-->ElasticSearch

2、springBoot默认支持两种技术和ES进行交互

1、Jest【需要导入使用】

利用JestClient和服务器的9200端口进行http通信

2、SpringData ElasticSearch【默认】

1)、客户端:Client节点信息: clusterNodes: clusterName

2)、ElasticsearchTemplate操作es

3)、编写ElasticsearchRepository子接口

1、Jest

1、注释SpringDataElasticSearch的依赖，并导入Jest【5.xx】的相关依赖

```

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

    <!--</dependency>-->
    <dependency>
        <groupId>io.searchbox</groupId>
        <artifactId>jest</artifactId>
        <version>5.3.3</version>
    </dependency>

```

2、修改配置文件application.yml

```
spring:
  elasticsearch:
    jest:
      uris: http://10.138.223.126:9200
```

3、创建 bean.Article

```
package com.wdjr.springboot.bean;

import io.searchbox.annotations.JestId;

public class Article {

    @JestId
    private Integer id;
    private String autor;
    private String title;
    private String content;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

4、运行程序

5、编写Jest Client的测试类

向wdjr-article中插入数据

```
@Test
public void contextLoads() {
    // 给Es中索引（保存）一个文档
    Article article = new Article();
    article.setId(1);
    article.setTitle("Effect Java");
    article.setAutor("Joshua Bloch");
    article.setContent("Hello world");
    // 构建一个索引功能
    Index index = new Index.Builder(article).index("cuzz").type("article").build();

    try {
        //执行
        jestClient.execute(index);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

<http://10.138.223.126:9200/cuzz/article/1>

Send

Preview

Add to collection

Body

Cookies (1)

Headers (3)

STATUS

200 OK

TIME

23 ms

Pretty

Raw

Preview

🖨

🔍

JSON

XML

```
1 {
2   "_index": "cuzz",
3   "_type": "article",
4   "_id": "1",
5   "_version": 1,
6   "found": true,
7   "_source": {
8     "id": 1,
9     "autor": "Joshua Bloch",
10    "title": "Effect Java",
11    "content": "Hello World"
12  }
13 }
```

查询数据

```
@Test
public void search(){
    // 查询表达式
    String json = "{\n" +
```

```

        "        \"query\" : {\n" +
        "            \"match\" : {\n" +
        "                \"content\" : \"Hello\"\n" +
        "            }\n" +
        "        }\n" +
        "    }";
// 构建搜索操作
Search search = new Search.Builder(json).addIndex("cuzz").addType("article").build();

// 执行
try {
    SearchResult result = jestClient.execute(search);
    System.out.println(result.getJsonString());
} catch (IOException e) {
    e.printStackTrace();
}
}

```

2、SpringData-Elastic

1、下载对应版本的ElasticSearch

如果版本不适配，会报错，解决方案：升级SpringBoot版本，或者安装合适的ES

spring data elasticsearch	elasticsearch
3.1.x	6.2.2
3.0.x	5.5.0
2.1.x	2.4.0
2.0.x	2.2.0
1.3.x	1.5.2

2、在Docker中安装适合版本的ES【2.4.6】

```

docker pull elasticsearch:2.4.6
docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9201:9200 -p 9301:9300 --name ES02 id

```

3、编写配置文件

```

spring:
  data:
    elasticsearch:
      cluster-name: elasticsearch
      cluster-nodes: 10.138.223.126:9301

```

4、修改pom文件，把使用data-elasticsearch，把刚才注释删除

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

5、操作ElasticSearch有两种方式

- 1)、编写一个ElasticsearchRepository
- 2)、编写一个ElasticsearchTemplate

6、ElasticsearchRepository的操作

- 1)、新建一个bean/Book类, **注意:** @Document(indexName = "cuzz", type="book")

```
/**
 * @Author: cuzz
 * @Date: 2018/9/27 18:32
 * @Description:
 */
@Document(indexName = "cuzz", type="book")
@Data
public class Book {
    private Integer id;
    private String bookName;
    private String auto;

    public Book() {
        super();
    }

    public Book(Integer id, String bookName, String auto) {
        super();
        this.id = id;
        this.bookName = bookName;
        this.auto = auto;
    }
}
```

- 2)、新建一个repository/BookRepository

方法编写参考[官方文档](#)


```

/**
 * @Author: cuzz
 * @Date: 2018/9/27 18:33
 * @Description:
 */
public interface BookRepository extends ElasticsearchRepository<Book, Integer> {
    //自定义查询方法
    public List<Book> findByBookNameLike(String bookName);
}

```

3)、编写测试类

```

@Autowired
BookRepository bookRepository;
@Test
public void testSearch(){
    for (Book book : bookRepository.findByBookNameLike("Effect")) {
        System.out.println(book);
    }
}

```

十二、SpringBoot的任务

1、异步任务

先开启异步注解，添加@EnableAsync

```

@EnableAsync
@SpringBootApplication
public class Springboot12TaskApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot12TaskApplication.class, args);
    }
}

```

service，在方法上添加@Async

```

/**
 * @Author: cuzz
 * @Date: 2018/9/28 17:49
 * @Description:
 */
@Service
public class AsyncService {

    @Async
    public void hello() {
        try {
            Thread.sleep(3);
        }
    }
}

```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("处理数据中...");
    }
}
```

controller

```
/**
 * @Author: cuzz
 * @Date: 2018/9/28 17:51
 * @Description:
 */
@RestController
public class AsynController {

    @Autowired
    AsynService asynService;

    @GetMapping("/hello")
    public String hello() {
        asynService.hello();
        return "success";
    }
}
```

发现不会堵塞在这里，而是先返回success，异步处理hello请求

2、定时任务

项目开发中经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供TaskExecutor、TaskScheduler 接口。

cron表达式：

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12	, - * /
星期	0-7或SUN-SAT 0,7是SUN	, - * ? / L C #

含义：

特殊字符	代表含义
,	枚举
-	区间
*	任意
/	步长
?	日/星期冲突匹配
L	最后
W	工作日
C	和calendar联系后计算过的值
#	星期, 4#2, 第2个星期四

主要有@Scheduled注解, cron()方法

```
public @interface Scheduled {

    /**
     * A cron-like expression, extending the usual UN*X definition to include triggers
     * on the second as well as minute, hour, day of month, month and day of week.
     * <p>E.g. {@code "0 * * * * MON-FRI"} means once per minute on weekdays
     * (at the top of the minute - the 0th second).
     * @return an expression that can be parsed to a cron schedule
     * @see org.springframework.scheduling.support.CronSequenceGenerator
     */
    String cron() default "";
}
```

测试类

```
/**
 * @Author: cuzz
 * @Date: 2018/9/29 10:25
 * @Description:
 */
@Service
public class ScheduledService {

    // 表示周一到周六当秒为0时执行一次
    @Scheduled(cron = "0 * * * * MON-SAT")
    public void hello() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String date = sdf.format(new Date());
        System.out.println(date + " hello...");
    }
}
```

开启定时任务注解@EnableScheduling

```
@EnableAsync
@EnableScheduling
@SpringBootApplication
public class Springboot12TaskApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot12TaskApplication.class, args);
    }
}
```

测试

```
2018-09-29 10:48:00  hello...
2018-09-29 10:49:00  hello...
```

3、邮件任务

1、邮件发送需要引入spring-boot-starter-mail

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

2、Spring Boot 自动配置MailSenderAutoConfiguration

3、定义MailProperties内容，配置在application.properties中

```
spring.mail.username=214769277@qq.com
spring.mail.password=xxxxxxxxxxx
spring.mail.host=smtp.qq.com
spring.mail.properties.mail.smtp.ssl.enable=true
```

4、自动装配JavaMailSender

```
@Autowired
JavaMailSenderImpl mailSender;

@Test
public void contextLoads() {
    SimpleMailMessage message = new SimpleMailMessage();

    message.setSubject("Hello world");
    message.setText("text");

    message.setTo("cuzz1234@163.com");
}
```

```
message.setFrom("214769277@qq.com");

mailSender.send(message);

}
```

5、测试邮件发送

十三、SpringBoot的安全

Spring Security是针对Spring项目的安全框架，也是Spring Boot底层安全模块默认的技术选型。他可以实现强大的web安全控制。对于安全控制，我们仅需引入spring-boot-starter-security模块，进行少量的配置，即可实现强大的安全管理。

1、几个类

- WebSecurityConfigurerAdapter：自定义Security策略
- AuthenticationManagerBuilder：自定义认证策略
- @EnableWebSecurity：开启WebSecurity模式

2、基本概念

- 应用程序的两个主要区域是“认证”和“授权”（或者访问控制）。这两个主要区域是Spring Security 的两个目标。
- “认证”（Authentication），是建立一个他声明的主体的过程（一个“主体”一般是指用户，设备或一些可以在你的应用程序中执行动作的其他系统）。
- “授权”（Authorization）指确定一个主体是否允许在你的应用程序执行一个动作的过程。为了抵达需要授权的店，主体的身份已经有认证过程建立。
- 这个概念是通用的而不只在Spring Security中

3、步骤

- 引入SpringSecurity，由于版本问题，pom文件如下

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.cuzz</groupId>
  <artifactId>springboot-13-security</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>springboot-13-security</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.12.RELEASE</version>
```

```

        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>
        <thymeleaf-layout-dialect.version>2.3.0</thymeleaf-layout-
dialect.version>
        <!--<thymeleaf-extras-springsecurity4.version>3.0.2.RELEASE</thymeleaf-
extras-springsecurity4.version>-->
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.thymeleaf.extras/thymeleaf-
extras-springsecurity4 -->
        <!--<dependency>-->
            <!--<groupId>org.thymeleaf.extras</groupId>-->
            <!--<artifactId>thymeleaf-extras-springsecurity4</artifactId>-->
        <!--</dependency>-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

- [官方文档](#)
- [页面文件下载](#)
- application.properties, 刚登入需要设置密码

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
security.user.name=root
security.user.password=root
security.user.role=ADMIN
```

- 编写SpringSecurity配置了

```
/**
 * @Author: cuzz
 * @Date: 2018/9/29 12:51
 * @Description:
 */
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter{
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // super.configure(http);
        // 定制请求的授权规则
        http.authorizeRequests().antMatchers("/").permitAll()
            .antMatchers("/level1/**").hasRole("VIP1")
            .antMatchers("/level2/**").hasRole("VIP2")
            .antMatchers("/level3/**").hasRole("VIP3");
    }
}
```

发现现在只能访问首页, 其他页面拒绝访问

- 自定义角色

```
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter{

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // super.configure(http);
        // 定制请求的授权规则
        http.authorizeRequests().antMatchers("/").permitAll()
            .antMatchers("/level1/**").hasRole("VIP1")
            .antMatchers("/level2/**").hasRole("VIP2")
            .antMatchers("/level3/**").hasRole("VIP3");
        // 开启登入功能, 如果权限就来到登入页面
        http.formLogin();
    }

    // 定义认证规则
```

```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    // super.configure(auth);
    auth.inMemoryAuthentication()
        .withUser("cuzz").password("123456").roles("VIP1","VIP2")
        .and()
        .withUser("cuxx").password("123456").roles("VIP3");
}
}

```

- 开启注销功能

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    // super.configure(http);
    // 定制请求的授权规则
    http.authorizeRequests().antMatchers("/").permitAll()
        .antMatchers("/level1/**").hasRole("VIP1")
        .antMatchers("/level2/**").hasRole("VIP2")
        .antMatchers("/level3/**").hasRole("VIP3");
    // 开启登入功能, 如果权限就来到登入页面
    http.formLogin();
    // 开启注销功能, 成功注销后回到首页
    http.logout().logoutSuccessUrl("/");
}

```

在页面添加

```

<form th:action="@{/logout}" method="post">
    <input type="submit" value="注销"/>
</form>

```

4、流程

- 登陆/注销
 - HttpSecurity配置登陆、注销功能
- Thymeleaf提供的SpringSecurity标签支持
 - 需要引入thymeleaf-extras-springsecurity4
 - sec:authentication="name"获得当前用户的用户名
 - sec:authorize="hasRole('ADMIN')当前用户必须拥有ADMIN权限时才会显示标签内容
- remember me
 - 表单添加remember-me的checkbox
 - 配置启用remember-me功能
- CSRF (Cross-site request forgery) 跨站请求伪造
 - HttpSecurity启用csrf功能, 会为表单添加
 - csrf的值, 提交携带来预防CSRF

十四、SpringBoot的分布式

1、Dubbo简介

1. Dubbo是什么？

dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架（告别Web Service模式中的WSDL，以服务者与消费者的方式在dubbo上注册）

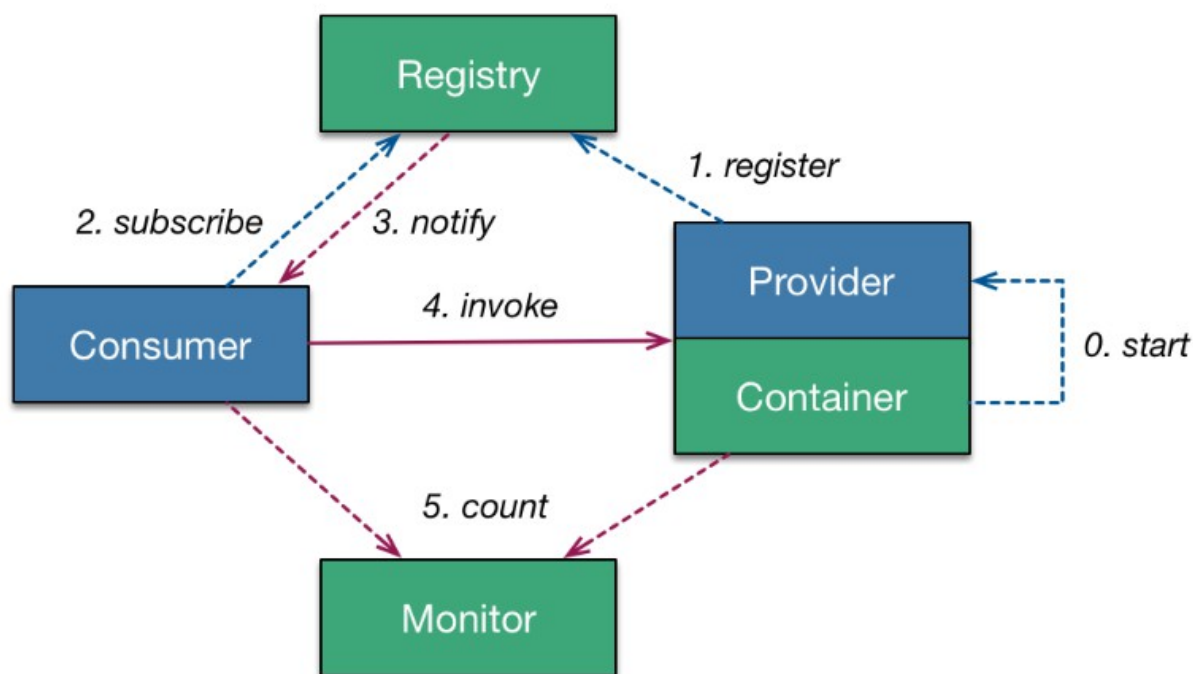
2. Dubbo能做什么？

- 1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。
- 2.软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。
- 3.服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

3、docker的原理

Dubbo Architecture

-----> init - - - - -> async ————> sync



调用关系说明：

0. 服务容器负责启动，加载，运行服务提供者。
1. 服务提供者在启动时，向注册中心注册自己提供的服务。
2. 服务消费者在启动时，向注册中心订阅自己所需的服务。

3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

2、Zookeeper

安装Zookeeper

```
#安装zookeeper镜像
docker pull registry.docker-cn.com/library/zookeeper
#运行zookeeper
docker run --name zk01 --restart always -d -p 2181:2181 bf5cbc9d5cac
```

3、Dubbo、Zookeeper整合

目的：完成**服务消费者**从注册中心查询调用**服务生产者**

1、将服务提供者注册到注册中心

1) 、引入dubbo和zkclient的相关依赖

```
<dependency>
  <groupId>com.alibaba.boot</groupId>
  <artifactId>dubbo-spring-boot-starter</artifactId>
  <version>0.1.0</version>
</dependency>

<dependency>
  <groupId>com.github.sgroschupf</groupId>
  <artifactId>zkclient</artifactId>
  <version>0.1</version>
</dependency>
```

2) 、配置service服务，新建service.TicketService 和service.TicketServiceImpl

```
public interface TicketService {
    public String getTicket();
}
```

```
package com.cuzz.ticket.service;

import com.alibaba.dubbo.config.annotation.Service;
import org.springframework.stereotype.Component;

/**
 * @Author: cuzz
 * @Date: 2018/9/30 12:28
 * @Description:
```

```

*/
@Component
@Service // 这个是dubbo @Service
public class TicketServiceImpl implements TicketService{

    @Override
    public String getTicket() {
        return "《大话西游》";
    }
}

```

3)、配置文件application.xml

```

# 名称
dubbo.application.name=provider-ticket
# 地址
dubbo.registry.address=zookeeper://10.138.223.126:2181
# 扫描哪些包
dubbo.scan.base-packages=com.cuzz.ticket.service

```

4)、启动服务提供者

2、启动服务消费者

1)、引入Dubbo和Zookeeper的依赖

```

<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.1.0</version>
</dependency>

<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>

```

2)、新建一个service.userService,并将TicketService的接口调用过来【全类名相同-包相同】

```

package com.cuzz.user.service;

import com.alibaba.dubbo.config.annotation.Reference;
import com.cuzz.ticket.service.TicketService;
import org.springframework.stereotype.Service;

/**
 * @Author: cuzz
 * @Date: 2018/9/30 12:32
 * @Description:

```

```

*/
@Service
public class UserService {

    @Reference
    TicketService ticketService;

    public void hello(){
        String ticket = ticketService.getTicket();
        System.out.println("您已经成功买票: "+ticket);
    }
}

```

3)、配置文件application.xml

```

# 名称
dubbo.application.name=consumer-user
# 地址
dubbo.registry.address=zookeeper://10.138.223.126:2181

```

4)、编写测试类测试

```

@Autowired
UserService userService;
@Test
public void contextLoads() {
    userService.hello();
}

```

结果展示:

```

2018-09-30 13:05:39.006 INFO 15220 --- [
2018-09-30 13:05:39.231 INFO 15220 --- [
您已经成功买票: 《大话西游》
2018-09-30 13:05:39.411 INFO 15220 --- [
2018-09-30 13:05:39.412 INFO 15220 --- [
2018-09-30 13:05:39.413 INFO 15220 --- [

```

4、SpringCloud

SpringCloud是一个分布式的整体解决方案，Spring Cloud为开发者提供了**在分布式系统（配置管理，服务器发现，熔断，路由，微代理，控制总线，一次性token，全局锁，leader选举，分布式session，集群状态）中快速构建的工具**，使用SpringCloud的开发者可以快速的驱动服务或者构建应用，同时能够和云平台资源进行对接。

SpringCloud分布式开发的五大常用组件

Eureka:找到

- 服务器发现 ——Netflix Eureka
- 客户端负载均衡——Netflix Ribbon
- 断路器——Netflix Hystrix 发现不了就及时断开
- 服务网关——Netflix Zuul 过滤请求
- 分布式配置——SpringCloud Config

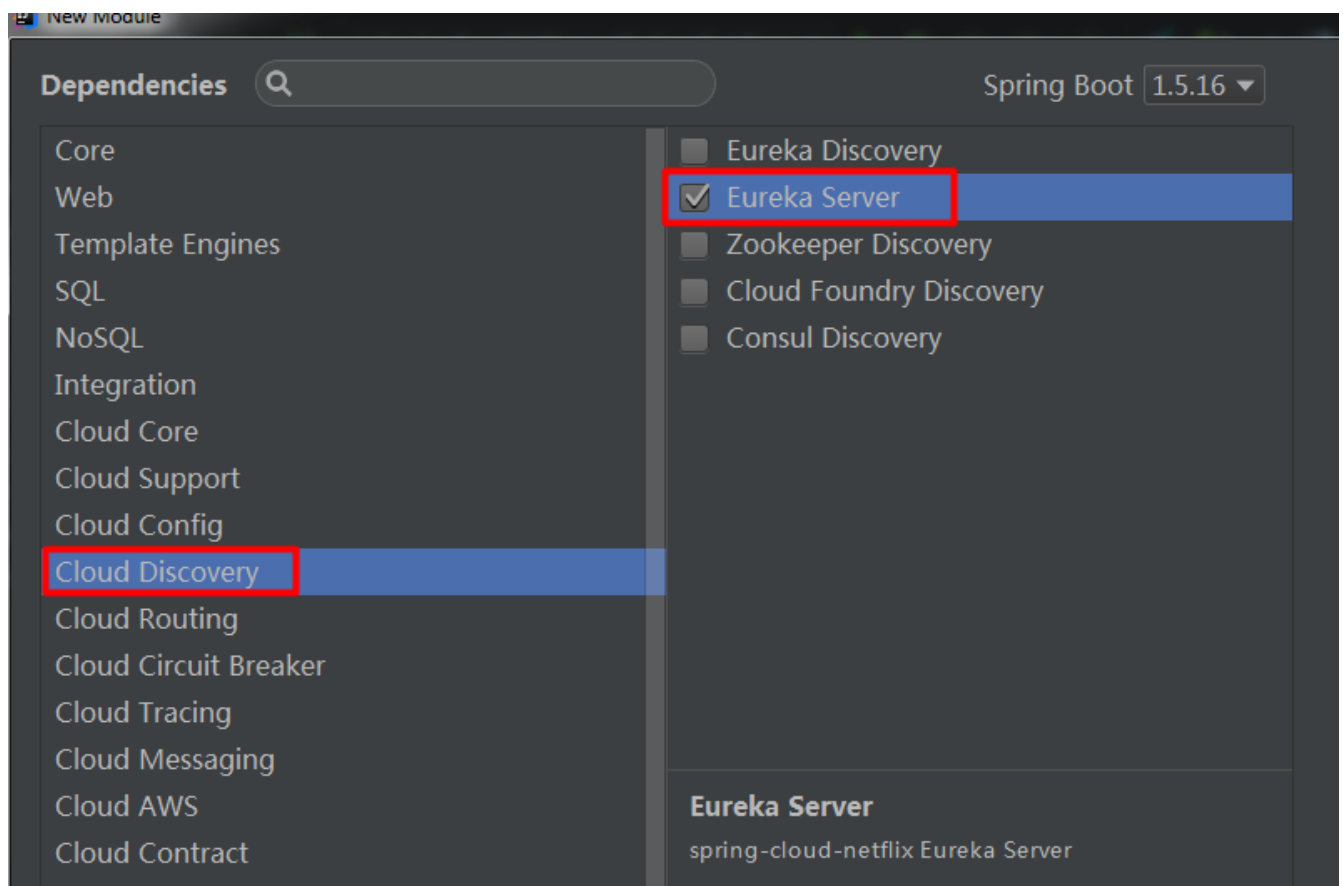
目的:

多个A服务调用多个B服务, 负载均衡

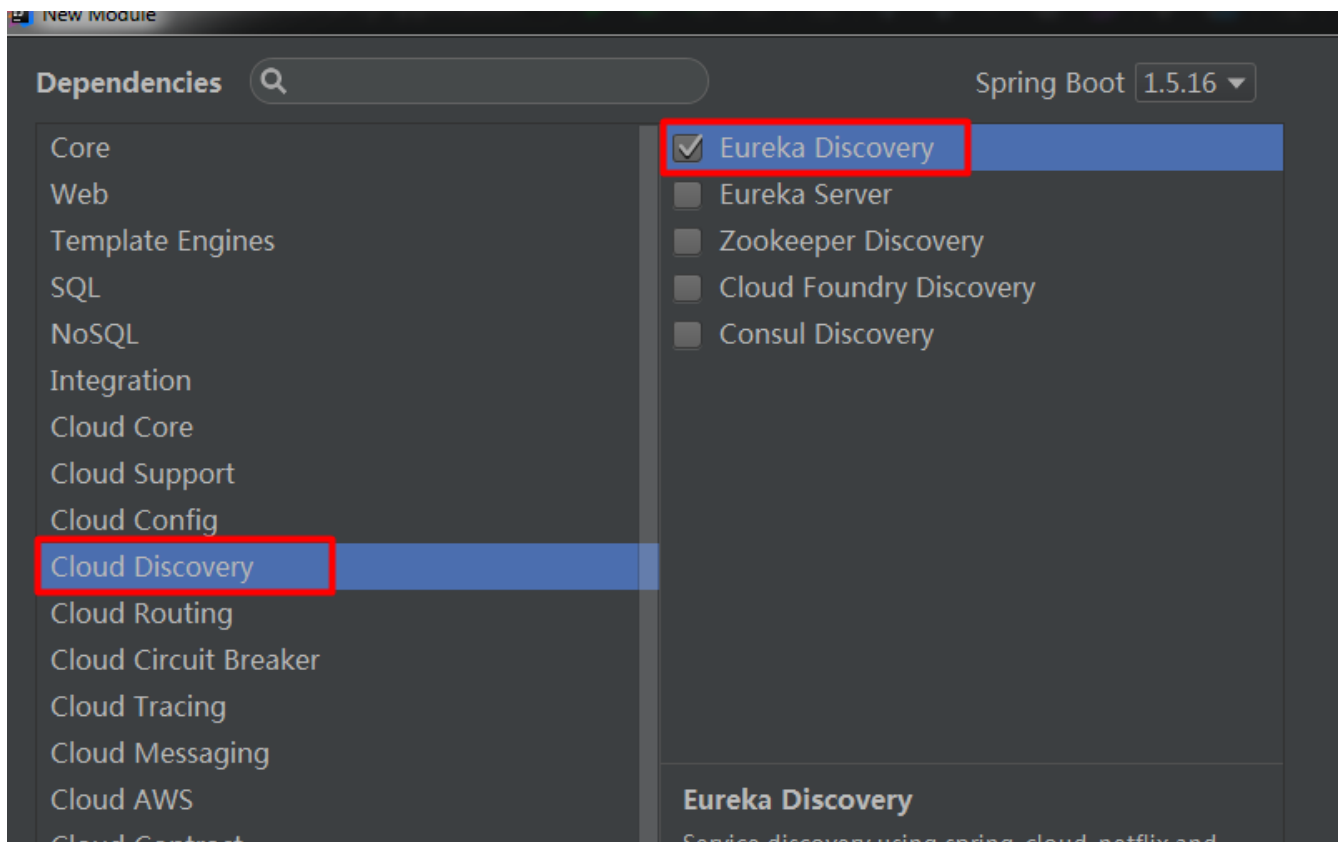
注册中心+服务提供者+服务消费者

创建项目

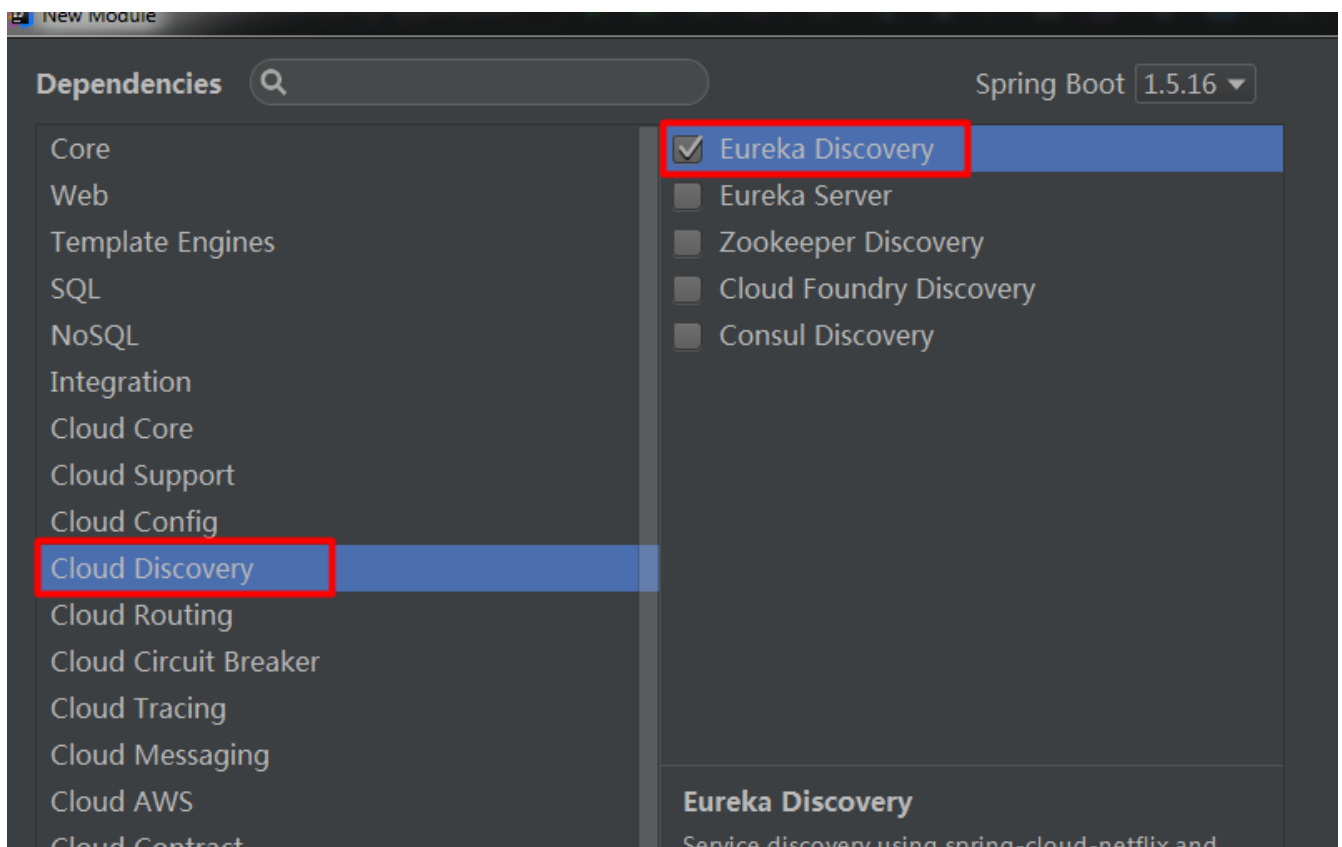
- eureka-server



- proviser-ticket



- consumer-user



1、注册中心 (eureka-server)

- 1、新建Spring项目，SpringBoot1.5+Eureka Server

2、编写application.yml

```
server:
  port: 8761
eureka:
  instance:
    hostname: eureka-server    #实例的主机名
  client:
    register-with-eureka: false #不把自己注册到eureka上
    fetch-registry: false      #不从eureka上来获取服务的注册信息
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

3、编写主程序

使用 `@EnableEurekaServer` 启用注册中心的功能

```
@EnableEurekaServer
@SpringBootApplication
public class Springboot14SpringcloudEurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot14SpringcloudEurekaServerApplication.class,
args);
    }
}
```

2、服务提供者 (provider-ticket)

1、新建Spring项目，SpringBoot1.5+Eureka Discovery

2、编写配置文件application.yml

```
server:
  port: 8001
spring:
  application:
    name: provider-ticket
eureka:
  instance:
    prefer-ip-address: true #注册是服务使用IP地址
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

3、创建一个售票的service

```

/**
 * @Author: cuzz
 * @Date: 2018/10/9 11:02
 * @Description:
 */
@Service
public class TicketService {
    public String getTicket() {
        return "《大话西游》";
    }
}

```

4、创建一个用于访问的controller

```

/**
 * @Author: cuzz
 * @Date: 2018/10/9 11:04
 * @Description:
 */
@RestController
public class TicketController {

    @Autowired
    TicketService ticketService;

    @GetMapping("/ticket")
    public String getTicket() {
        return ticketService.getTicket();
    }
}

```

5、完毕

3、服务消费者 (consumer-user)

1、新建Spring项目，SpringBoot1.5+Eureka Discovery

2、编写application.yml文件

```

spring:
  application:
    name: consumer-user
server:
  port: 8200
eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

```


3、编写一个controller

```
@RestController
public class UserController {

    @Autowired
    RestTemplate restTemplate;
    @GetMapping("/buy")
    public String buyTicket(String name){
        String s = restTemplate.getForObject("http://PROVIDER-TICKET/ticket", String.class);
        return name+"购买了"+" "+s;
    }
}
```

4、编写主程序

```
@EnabledDiscoveryClient // 开启发现服务
@SpringBootApplication
public class Springboot14SpringcloudConsumerUserApplication {

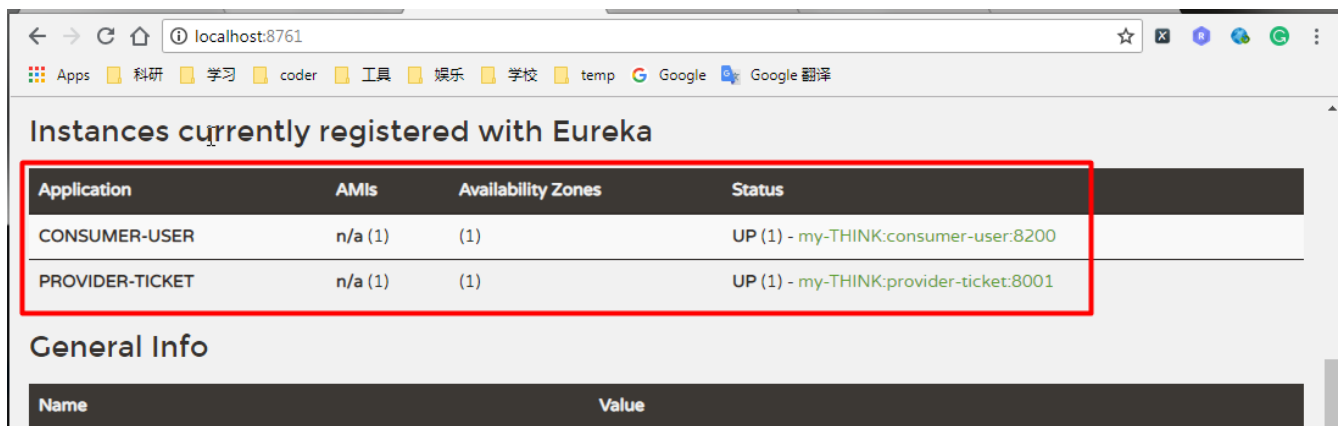
    public static void main(String[] args) {
        SpringApplication.run(Springboot14SpringcloudConsumerUserApplication.class,
args);
    }

    @LoadBalanced //使用负载均衡机制
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

5、完毕

4、测试

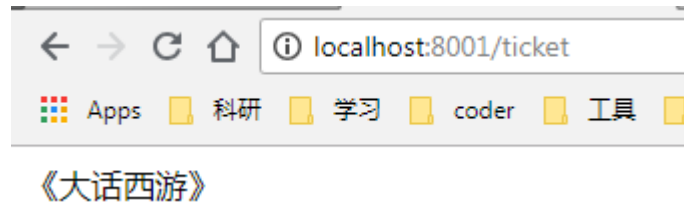
1、运行Eureka-server, provider-ticket【8002执行】(端口改为8001打成jar包, 执行), consumer-user



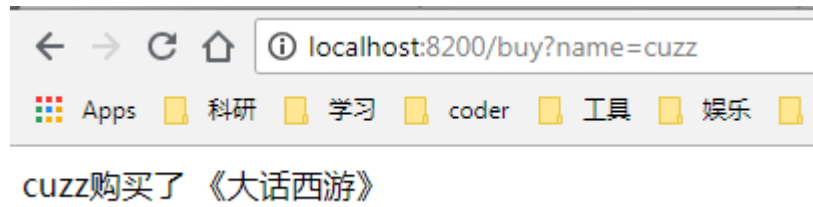
Application	AMIs	Availability Zones	Status
CONSUMER-USER	n/a (1)	(1)	UP (1) - my-THINK:consumer-user:8200
PROVIDER-TICKET	n/a (1)	(1)	UP (1) - my-THINK:provider-ticket:8001

Name	Value
------	-------

2、provider-ticket



3、consumer-user



访问是以负载均衡的方式，所以每次都是 8001 。8002.轮询访问S

十五、Spring Boot与开发热部署

十六、SpringBoot的监管
