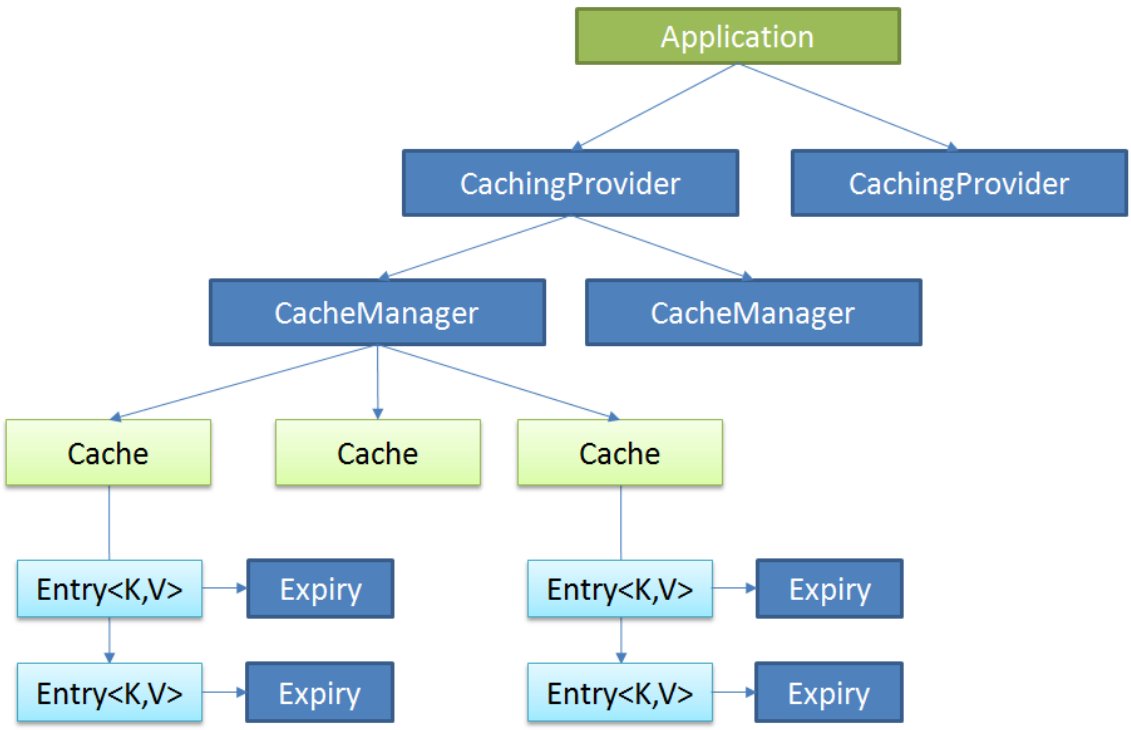
一、Spring Boot 之缓存

1、JSR107

> Java Caching定义了5个核心接口，分别是CachingProvider, CacheManager, Cache, Entry 和 Expiry。
>
> - 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仅被一个CachingProvider所拥有。
> - Cache是一个类似Map的数据结构并临时存储以Key为索引的值。一个Cache仅被一个CacheManager所拥有。
> - Entry是一个存储在Cache中的key-value对。
> - Expiry 每一个存储在Cache中的条目有一个定义的有效期。一旦超过这个时间，条目为过期的状态。一旦过期，条目将不可访问、更新和删除。缓存有效期可以通过ExpiryPolicy设置。

缓存的类图



2、Spring的缓存抽象

Spring从3.1开始定义了org.springframework.cache.Cache和org.springframework.cache.CacheManager接口来统一不同的缓存技术；

并支持使用JCache（JSR-107）注解简化我们开发

- Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；

- Cache接口下Spring提供了各种xxxCache的实现；如RedisCache，EhCacheCache , ConcurrentMapCache等；
- 每次调用需要缓存功能的方法时，Spring会检查检查指定参数的指定的目标方法是否已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。
- 使用Spring缓存抽象时我们需要关注以下两点；

  1、确定方法需要被缓存以及他们的缓存策略

  2、从缓存中读取之前缓存存储的数据

3.几个重要概念&缓存注解

- 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 :主要针对方法配置，能够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 .
- @EnableCaching :开启基于注解的缓存
- keyGenerator :缓存数据时key生成策略
- serialize :缓存数据时value序列化策略

@Cacheable/@CachePut/@CacheEvict 主要的参数

value	**缓存的名称，在 spring 配置文件中定义，必须指定至少一个**	**例如： @Cacheable(value=”mycache”) 或者 @Cacheable(value={”cache1”,”cache2”}**
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @Cacheable(value=”testcache”,key=”#userName”)
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存，在调用方法之前之后都能判断 .	例如： @Cacheable(value=”testcache”,condition=”#userName.length()>2”)
allEntrie((@CacheEvict))	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： @CachEvict(value=”testcache”,allEntries=true) P
beforeInvocation	是否在方法执行前就清空，缺省为 false	例如：@CachEvict(value=”testcache”，beforeInvocation=true)
unless	用于否决缓存的，不像 condition，该表达式只在方法执行之后判断，此时可以拿到返回值result进行判断。条件为true不会缓存，fasle才缓存	例如： @Cacheable(value=”testcache”,unless=”#result == null”) m

4.缓存机制的原理

1）、自动配置类：CacheAutoConfiguration

导入CacheConfigurationImportSelector这个类：

```
static class CacheConfigurationImportSelector implements ImportSelector {

        @Override
        public String[] selectImports(AnnotationMetadata importingClassMetadata) {
            CacheType[] types = CacheType.values();
            String[] imports = new String[types.length];
            for (int i = 0; i < types.length; i++) {
                imports[i] = CacheConfigurations.getConfigurationClass(types[i]);
            }
            return imports;
        }

    }
```

2）、上面代码导入缓存的配置类，共有如下几种

- org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration
- org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration
- org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration
- org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration
- org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration
- org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration
- org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration
- org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration
- org.springframework.boot.autoconfigure.cache.GuavaCacheConfiguration
- org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration
- org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration

3）、那些配置类会默认会生效呢？SimpleCacheConfiguration

4）、给容器中注册了一个CacheManager:ConcurrentMapCacheManager

5）、可以获取和创建ConcurrentMapCache类型的缓存组件：他的作用将数据保存在ConcurrentMap中

运行流程

1）、方法运行之前，先去查询Cache（缓存组件），暗战cacheNames指定的名字获取

（CacheManager先获取相应的缓存），第一次获取缓存如果没有Cache组件会自动创建

2）、去Cache中查找缓存的内容，使用一个key，默认的方法就是参数；

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

3）、没有查到缓存就调用目标方法

4）、将目标方法返回的结构，放进缓存中

5.整合redis

1）、引入spring-boot-starter-data-redis

2）、application.yml配置redis连接地址

3）、使用RestTemplate操作redis

```
redisTemplate.opsForValue();//操作字符串
redisTemplate.opsForHash();//操作hash
redisTemplate.opsForList();//操作list
redisTemplate.opsForSet();//操作set
redisTemplate.opsForZSet();//操作有序set
```

4）、配置缓存、CacheManagerCustomizers

5）、测试使用缓存、切换缓存、 CompositeCacheManager

> - 引入redis的starter,容器中保存的是RedisCacheManager，
> - RedisCacheManager帮我们创建RedisCache来作为缓存组件，RedisCache通过操作Redis缓存数据
> - 默认创建RedisCacheManager操作redis的时候使用的死RedisTemplate<Object,Object>。默认使用jdk的序列化机制
> - 自定义CacheManager

二、SpringBoot之消息队列

1、概述

1.大多应用中，可通过消息服务中间件来提升系统异步通信、扩展解耦能力

2.消息服务中两个重要概念：

```
    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目的地。

3.消息队列主要有两种形式的目的地

1.队列（queue）：点对点消息通信（point-to-point）

2.主题（topic）：发布（publish）/订阅（subscribe）消息通信

4.点对点式：

-消息发送者发送消息，消息代理将其放入一个队列中，消息接收者从队列中获取消息内容，消息读取后被移出队列

-消息只有唯一的发送者和接受者，但并不是说只能有一个接收者

5.发布订阅式：

-发送者（发布者）发送消息到主题，多个接收者（订阅者）监听（订阅）这个主题，那么就会在消息到达时同时收到消息

6.JMS（Java Message Service）JAVA消息服务：

–基于JVM消息代理的规范。ActiveMQ、HornetMQ是JMS实现

7.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
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型： （1）、Peer-2-Peer （2）、Pub/sub	提供了五种消息模型： （1）、direct exchange （2）、fanout exchange （3）、topic change （4）、headers exchange （5）、system exchange 本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	多种消息类型： TextMessage MapMessage BytesMessage StreamMessage ObjectMessage Message（只有消息头和属性）	byte[] 当实际应用时，有复杂的消息，可以将消息序列化后发送。
综合评价	JMS 定义了JAVA API层面的标准；在java体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差；	AMQP定义了wire-level层的协议标准；天然具有跨平台、跨语言特性。

8.Spring支持

–spring-jms提供了对JMS的支持

–spring-rabbit提供了对AMQP的支持

–需要ConnectionFactory的实现来连接消息代理

–提供JmsTemplate、RabbitTemplate来发送消息

–@JmsListener（JMS）、@RabbitListener（AMQP）注解在方法上监听消息代理发布的消息

–@EnableJms、@EnableRabbit开启支持

9.Spring Boot自动配置

–JmsAutoConfiguration

RabbitAutoConfiguration

2、RabbitMQ

RabbitMQ简介：

RabbitMQ是一个由erlang开发的AMQP(Advanved Message Queue Protocol)的开源实现。

核心概念

Message

消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

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有4种类型：direct(默认)，fanout, topic, 和headers，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其取走。

Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

Exchange 和Queue的绑定可以是多对多的关系。

Connection

网络连接，比如一个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。信道是建立在真实的TCP连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

Consumer

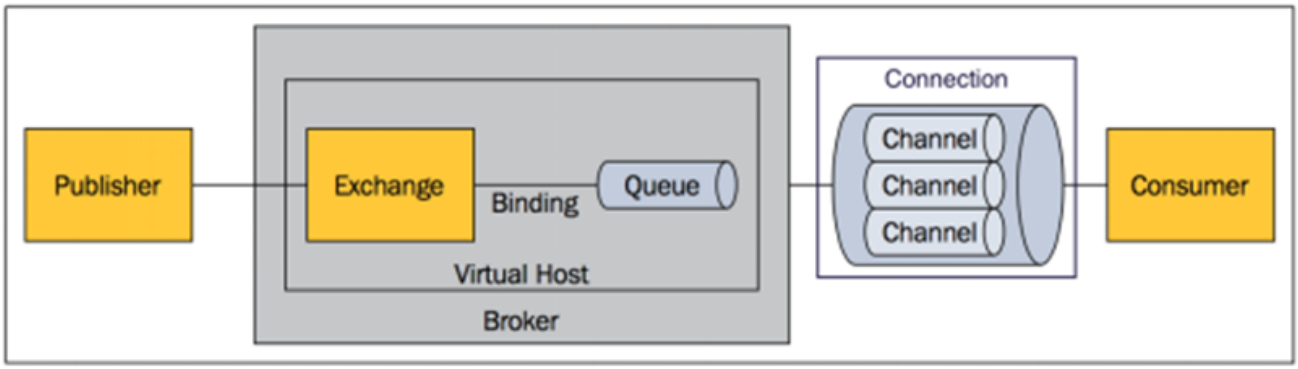
消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

Virtual Host

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

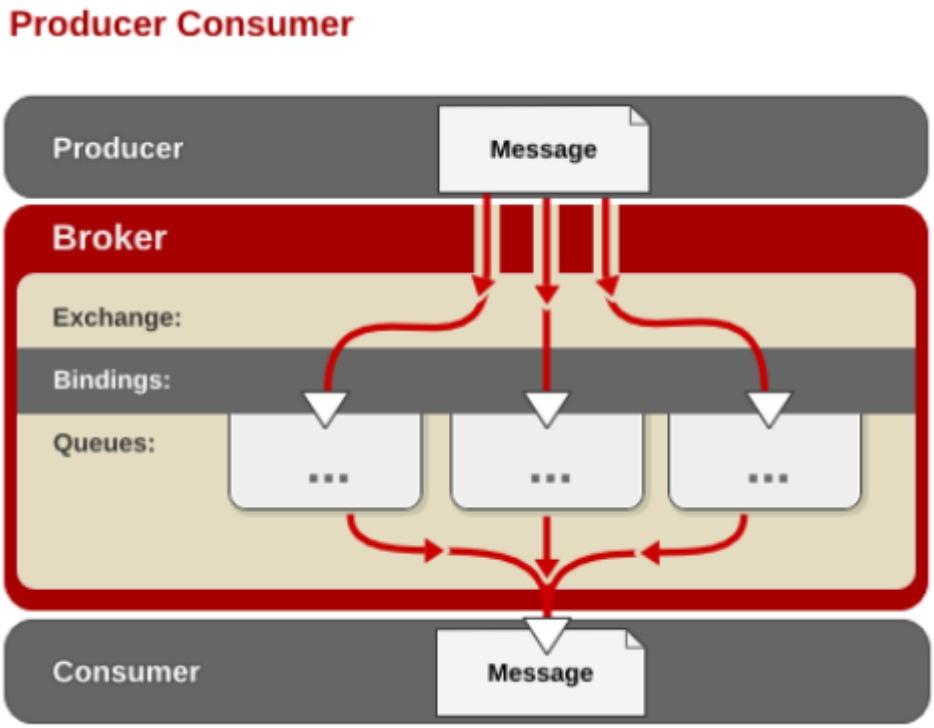
表示消息队列服务器实体



RabbitMQ运行机制

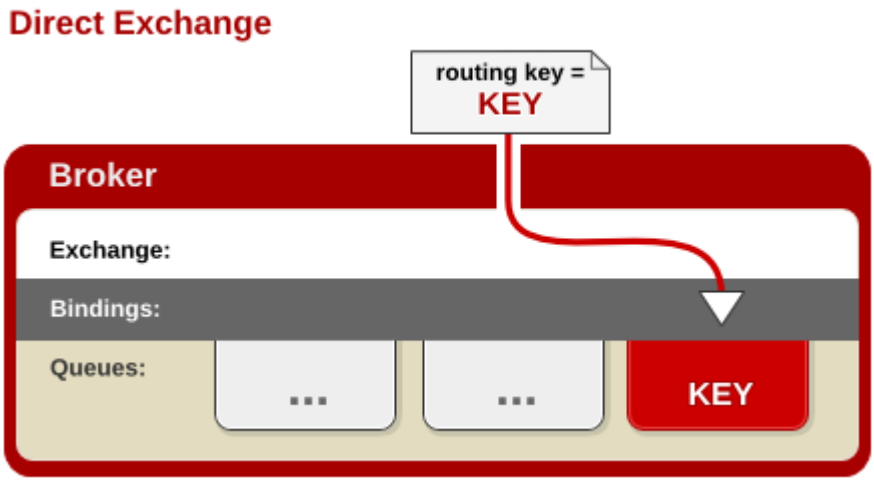
AMQP 中的消息路由

AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列。

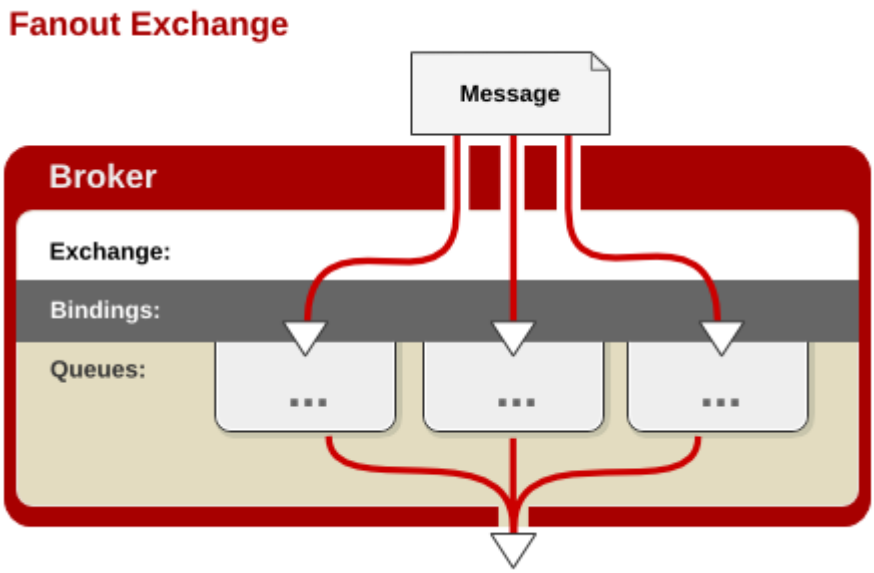


Exchange 类型

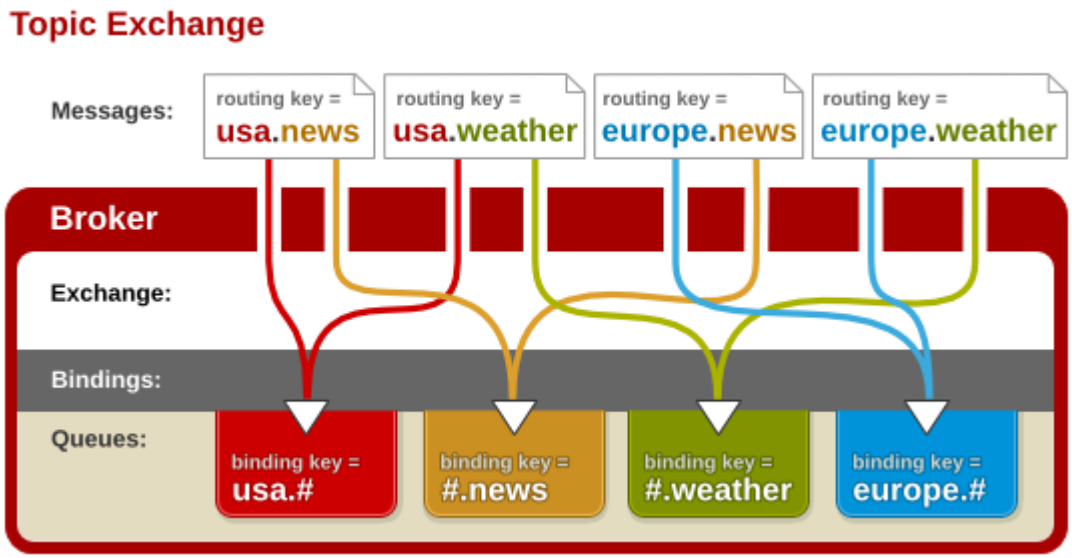
Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键， headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：



消息中的路由键（routing key）如果和 Binding 中的 binding key 一致， 交换器就将消息发到对应的队列中。路由键与队列名完全匹配，如果一个队列绑定到交换机要求路由键为“dog”，则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等。它是完全匹配、单播的模式。



每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

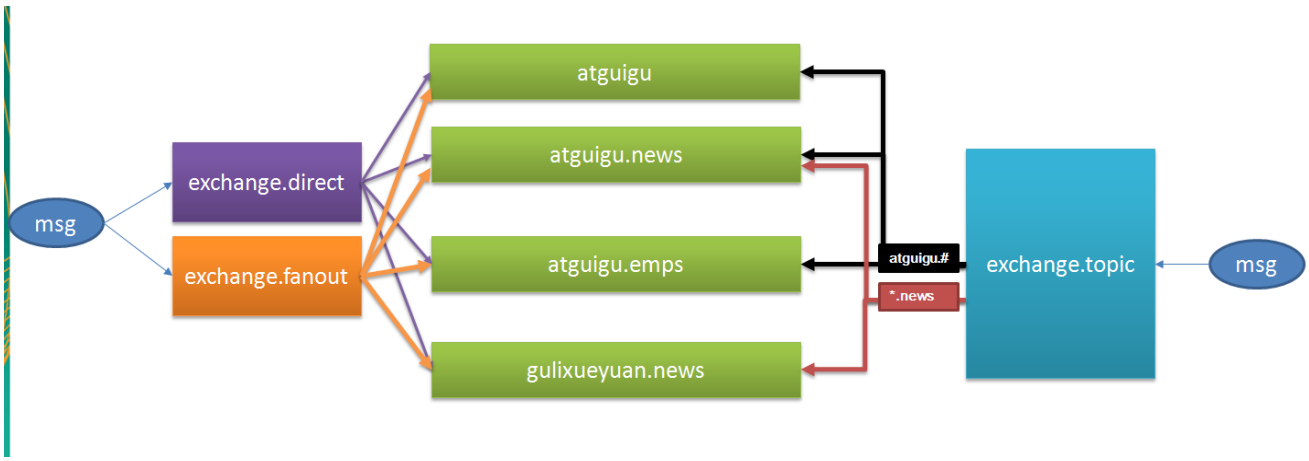


topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。*#匹配0个或多个单词*，匹配一个单词。

RabbitMQ整合

> 1）、引入 spring-boot-starter-amqp
>
> 2）、application.yml配置
>
> 3）、测试RabbitMQ

```
1.AmqpAdmin：管理组件

2.RabbitTemplate：消息发送处理组件
```



自动配置

1）、RabbitAutoConfiguration

2）、有自动配置连接工厂ConnectionFactory

3）、RabbitProperties封装了RabbitMQ的配置

4）、RabbitTemplate:给RabbitMQ发送和接受消息

5）、AmqpAdmin：RabbitMQ系统管理功能组件，创建和删除Queue，Exchange，Binding

```
    @Autowired
    AmqpAdmin amqpAdmin;

    public void createExchange(){
        //凡是以declare开始的都是创建。
        amqpAdmin.declareExchange(new DirectExchange(""));
    }
```

6）、@EnableRabbit+@RabbitListenner：开启基于注解的RabbitMQ模式，监听消息队列

```
    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     * 单播（点对点）
     */
    @Test
    public void contextLoad(){
        //message需要自己构建一个；定义消息体内容和消息头
        Map<String,Object> map = new HashMap<>();
        map.put("msg","");
        map.put("data","");
//        rabbitTemplate.send("exchange.direct","hello",map);
        //object默认当成消息体，只需要传入要发送的对象，自动化序列发给rabbitmq
        rabbitTemplate.convertAndSend("exchange.direct","hello",map);
    }

    @Test
    public void receive(){
        //接受消息
        Object o = rabbitTemplate.receiveAndConvert("QueueName");

    }
```

如何将数据序列化成Json,需要注入自己的MessageConverter

```
@Configuration
public class MyAMQPConfig {

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

3.SpringBoot之检索

概念

我们的应用经常需要添加检索功能，开源的 ElasticSearch 是目前全文搜索引擎的首选。他可以快速的存储、搜索和分析海量数据。Spring Boot通过整合Spring Data ElasticSearch为我们提供了非常便捷的检索功能支持；**Elasticsearch**是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard（分片）的方式保证数据安全，并且提供自动resharding的功能，github等大型的站点也是采用了ElasticSearch作为其搜索服务。

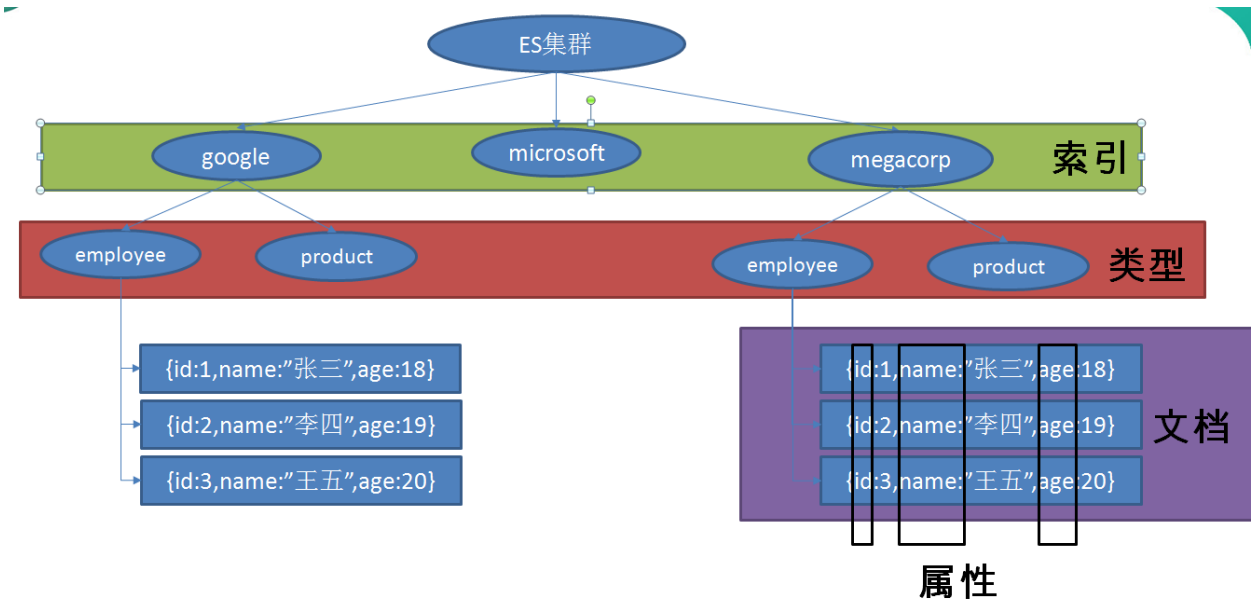
- 以 员工文档 的形式存储为例：一个文档代表一个员工数据。存储数据到 ElasticSearch 的行为叫做 索引 ，但在索引一个文档之前，需要确定将文档存储在哪里。
- 一个 ElasticSearch 集群可以 包含多个 索引 ，相应的每个索引可以包含多个 类型 。 这些不同的类型存储着多个 文档 ，每个文档又有 多个 属性 。
- 类似关系：

  -索引-数据库

  -类型-表

  -文档-表中的记录

  -属性-列



ElasticSearch安装（使用Docker）

1）、安装elasticsearch镜像，使用国内镜像加速

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

2）、启动，需要配置占用内存大小，否则可能会出现问题

```
docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 --name ES01
097d037f8ff8
```

3）、测试，打开浏览器，输入 localhost:9200

```
{
    "name": "6bqyUpK",
    "cluster_name": "elasticsearch",
    "cluster_uuid": "RGsYd_O4R4G5hcwnsXO4rQ",
    "version": {
        "number": "5.6.11",
        "build_hash": "bc3eef4",
        "build_date": "2018-08-16T15:25:17.293Z",
        "build_snapshot": false,
        "lucene_version": "6.6.1"
    },
    "tagline": "You Know, for Search"
}
```

整合ElasticSearch测试

1）、引入依赖

```
    <!-- 注意版本适配问题 https://github.com/spring-projects/spring-data-elasticsearch-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
    </dependency>
```

```
<!-- https://mvnrepository.com/artifact/io.searchbox/jest -->
<dependency>
    <groupId>io.searchbox</groupId>
    <artifactId>jest</artifactId>
    <version>6.3.1</version>
</dependency>
```

2）、application.yml配置 (分别对应Jest，SpringData)

```
spring.elasticsearch.jest.uris=http://localhost:9200
```

```
spring.data.elasticsearch.cluster-name=elasticsearch
spring.data.elasticsearch.cluster-nodes=http://localhost:9301
```

3）、测试

Jest

```
    @Autowired
    private JestClient jestClient;

     @Test
    public void contextLoad(){
        //给ES中索引保存一个文档
        Article article = new Article();
        //setter
        //构建一个索引功能
        Index index = new Index.Builder(article).index("jcohy").type("news").build();
        try {
            //执行
            jestClient.execute(index);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //测试搜索
    @Test
    public void search(){
        String json = "";
        //构建搜索功能
        Search search = new
Search.Builder(json).addIndex("jcohy").addType("news").build();
        //执行
        try {
            SearchResult searchResult = jestClient.execute(search);
            System.out.println(searchResult.getJsonString());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

SpringData

编写ElasticSearchRepository

```
public interface BookRepository extends ElasticsearchRepository<Book, String> {

        List<Book> findByNameAndPrice(String name, Integer price);

        List<Book> findByNameOrPrice(String name, Integer price);

        Page<Book> findByName(String name,Pageable page);

        Page<Book> findByNameNot(String name,Pageable page);

        Page<Book> findByPriceBetween(int price,Pageable page);
```

```
    Page<Book> findByNameLike(String name,Pageable page);

    @Query("{\"bool\" : {\"must\" : {\"term\" : {\"message\" : \"?0\"}}}}")
    Page<Book> findByMessage(String message, Pageable pageable);
}
```

Book

```
@Document(indexName = "jcohy",type = "book")
public class Book {

    private Integer id;
    private String name;
    private String author;
    private Integer price;

    public Integer getPrice() {
        return price;
    }

    public void setPrice(Integer price) {
        this.price = price;
    }
    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("Book{");
        sb.append("id=").append(id);
        sb.append(", price='").append(price).append('\'');
```

```
        sb.append(", name='").append(name).append('\'');
        sb.append(", author='").append(author).append('\'');
        sb.append('}');
        return sb.toString();
    }
}
```

```
    @Autowired
    private BookRepository bookRepository;

    @Test
    public void test2(){
        Book book = new Book();
        book.setId(1);
        book.setName("jcohy");
        book.setAuthor("jcohy");
        bookRepository.index(book);
    }
```

ElasticSearch自动配置

SpringBoot默认支持两种技术来和ES交互

1）、Jest（默认不生效），需要导入jest的工具包（io.searchbox.client.JestClient） 2）、SpringData ElasticSearch 1）、Client节点信息：Client ClusterNodes；ClusterNames 2）、ElasticSearchTemplate操作ES

```
3）、编写一个ElasticSearchRepository的子接口来操作ES
```

文档链接

4.SpringBoot之异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在处理与第三方系统交互的时候，容易造成响应迟缓的情况，之前大部分都是使用多线程来完成此类任务，其实，在Spring 3.x之后，就已经内置了@Async来完美解决这个问题。

两个注解：@EnableAysnc、@Aysnc

在启动类上开启异步注解功能：

```
@EnableAsync
@SpringBootApplication
public class SpringBootStudyApplication {
    public static void main(String[] args) {
        // Spring应用启动起来
        SpringApplication.run(SpringBootStudyApplication.class,args);
    }
}
```

Controller：

```
@RestController
public class AsyncController {

    @Autowired
    private AsyncService asyncService;

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

Service：

```
@Service
public class AsyncService {

    @Async
    public void hello(){
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("数据处理中");
    }
}
```

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。

两个注解：@EnableScheduling、@Scheduled

开启基于注解的定时任务:

```
@@EnableScheduling
@SpringBootApplication
public class SpringBootStudyApplication {
    public static void main(String[] args) {
        // Spring应用启动起来
        SpringApplication.run(SpringBootStudyApplication.class,args);
    }
}
```

Service:

```
@Service
public class ScheduledService {

    /**
     * second(秒)  minute(分), hour(时), day of month(日), month(月),and day of
week.
     * 0 * * * * MON-FRI
     * 【0 0/5 14,18 * * ?】每天14点整和18点整，每隔5分钟执行一次
     * 【0 15 10 ? * 1-6】每个月的周一至周六10：15分执行一次
     * 【0 0 2 ? * 6L】每个月的最后一个周六凌晨2点执行一次
     * 【0 0 2 LW * ?】每个月的最后一个工作日凌晨2点执行一次
     * 【0 0 2-4 ? * 1#1】每个月的第一个周一凌晨2点到4点期间，每个整点都执行一次
     */
//    @Scheduled(cron = "0 * * * * MON-FRI")
//    @Scheduled(cron = "0,1,2,3,4 * * * * MON-FRI")
//    @Scheduled(cron = "0-4 * * * * MON-FRI")
    @Scheduled(cron = "0/4 * * * * MON-FRI")//每4秒执行一次
    public void hello(){
        System.out.println("hello...");
    }
}
```

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 #

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

邮件任务

SpringBoot整合邮件任务

1）、引入Spring-boot-starter-mail

2）、自动配置：

MailSenderAutoConfiguration ，MailProperties

3）、配置application.yml

```
spring.mail.username=534096094@qq.com
spring.mail.password=gtstkoszjelabijb
spring.mail.host=smtp.qq.com
spring.mail.properties.mail.smtp.ssl.enable=true
```

4）、自动装配JavaMailSender

5）、测试：

```
public class MailTest {

    @Autowired
    private JavaMailSenderImpl javaMailSender;

    @Test
    public void test(){
        SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
        //邮件设置
        simpleMailMessage.setSubject("通知：xxx");
        simpleMailMessage.setText("内容");
        simpleMailMessage.setTo("");
        simpleMailMessage.setFrom("");
        javaMailSender.send(simpleMailMessage);
```

```
    }

    @Test
    public void test02() throws Exception {
        //1.创建一个复杂的消息邮件
        MimeMessage message =javaMailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message,true);
        //邮件设置
        helper.setSubject("通知：xxx");
        helper.setText("内容");
        helper.setTo("");
        helper.setFrom("");
        //上传文件
        helper.addAttachment("1.jpg",new File(""));
        javaMailSender.send(message);
    }
}
```

5.SpringBoot之安全

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。 几个类：

WebSecurityConfigurerAdapter：自定义Security策略

AuthenticationManagerBuilder：自定义认证策略

@EnableWebSecurity：开启WebSecurity模式

- 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。

SpringSecurity整合

1）、引入SpringSecurity

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

2）、编写SpringSecurity的配置

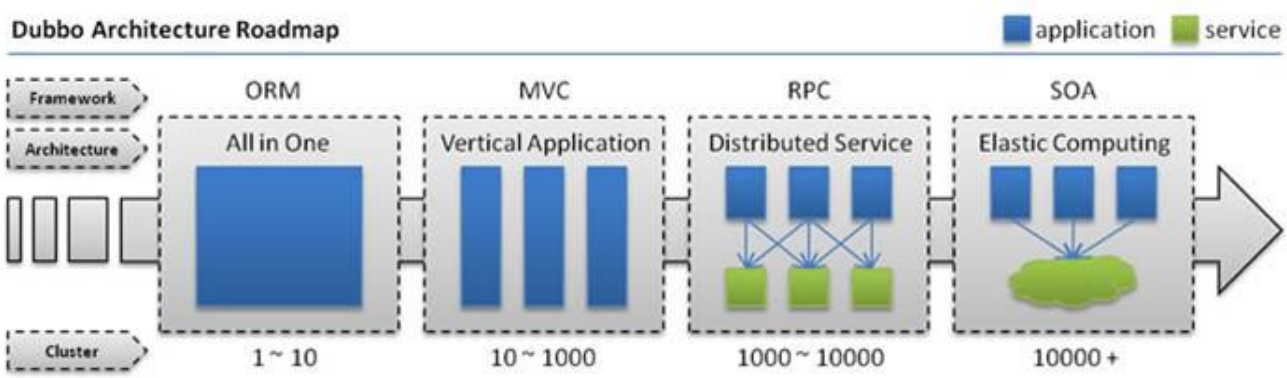
可参考官方文档

```
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter{

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //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");
        //开启自动配置的登录功能，如果没有权限，就会来到登录页面
        http.formLogin();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().withUser("zhangsan")
                .password("123456")
                .roles("VIP1","VIP2")
                .and()
                .withUser("lisi")
                .password("123456")
                .roles("VIP1","VIP3")
                .and()
                .withUser("lisi")
                .password("123456")
                .roles("VIP2","VIP3");
    }
}
```

6.SpringBoot之分布式

在分布式系统中，国内常用zookeeper+dubbo组合，而Spring Boot推荐使用全栈的Spring，Spring Boot+Spring Cloud

分布式系统：



- 单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

- 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

- 分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

- 流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

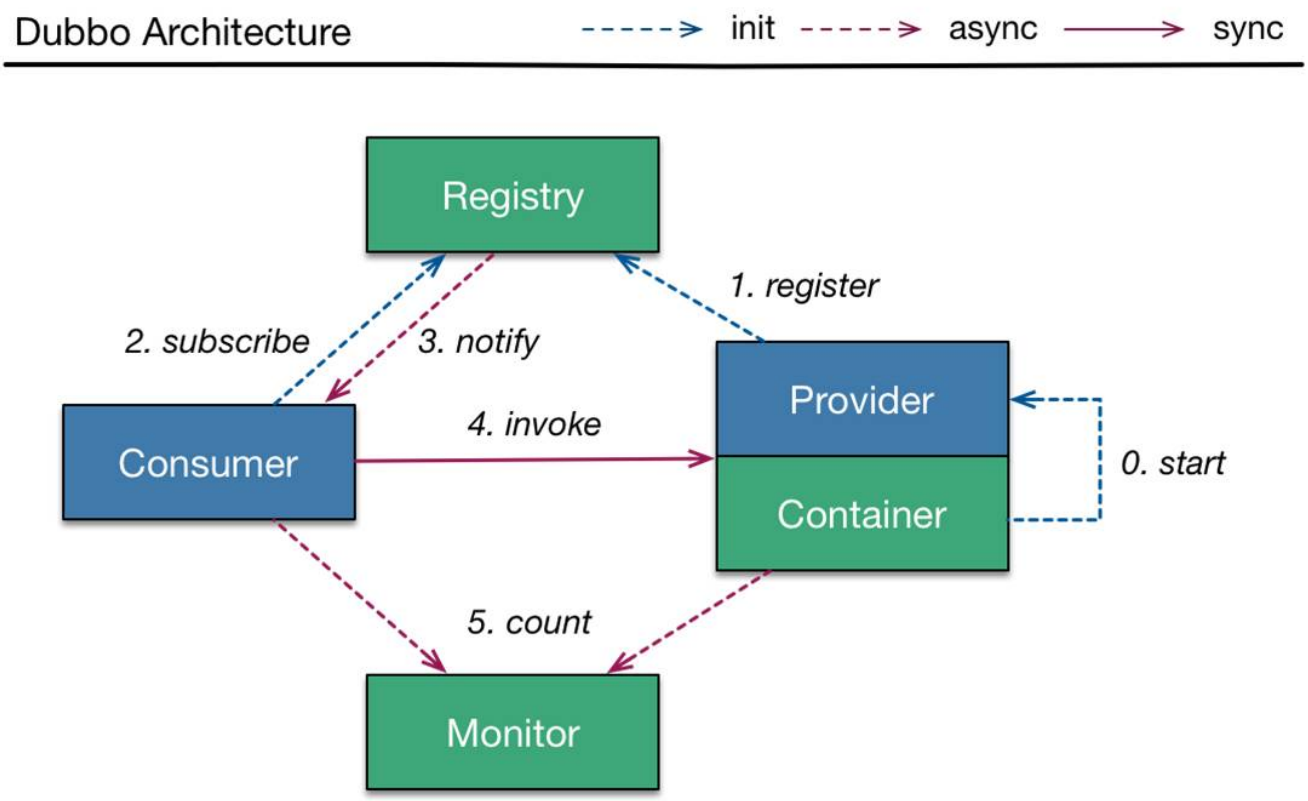
Zookeeper和Dubbo

ZooKeeper

> zookeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

Dubbo

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。



SpringBoot整合

dubbo和zookeeper结合

1）、引入dubbo和zkclient依赖

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

<!--引入zookeeper的客户端工具-->
<!-- https://mvnrepository.com/artifact/com.github.sgroschupf/zkclient -->
<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>
```

2）、配置dubbo的扫描包和注册中心地址

```
dubbo.application.name=provider-ticket

dubbo.registry.address=zookeeper://118.24.44.169:2181

dubbo.scan.base-packages=com.atguigu.ticket.service
```

3）、使用@Service发布服务

4）、使用@Reference引用服务

SpringCloud

Spring Cloud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，服务发现，熔断，路由，微代理，控制总线，一次性token，全局琐，leader选举，分布式session，集群状态）中快速构建的工具，使用Spring Cloud的开发者可以快速的启动服务或构建应用、同时能够快速和云平台资源进行对接。

SpringCloud分布式开发五大常用组件

- 服务发现——Netflix Eureka
- 客服端负载均衡——Netflix Ribbon
- 断路器——Netflix Hystrix
- 服务网关——Netflix Zuul
- 分布式配置——Spring Cloud

1）、引入Eureka-server

```
<dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

2）、配置Eureka信息

```
server:
  port: 8761
eureka:
  instance:
    hostname: eureka-server  # eureka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）、使用@EnableEurekaServer注解开启服务

4）、测试

localhost:8761 可以查看Eureka服务注册相关信息

5）、服务提供者配置

```yaml
server:
  port: 8002
spring:
  application:
    name: provider-ticket

eureka:
  instance:
    prefer-ip-address: true # 注册服务的时候使用服务的ip地址
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

6）、服务消费者配置

@EnableDiscoveryClient :开启发现服务功能

添加RestTemplate

```
@LoadBalanced
@Bean
public RestTemplate restTemplate(){
    return new RestTemplate
}
```

```
@Autowired
RestTemplate restTemplate;
public String buTicket(String name){
    String forObject = restTemplate.getForObject("http://PROVIDER-
RICKET/ticket",String.class);
    return name+"购买了"+forObject;
}
```

7.SpringBoot之热部署

在开发中我们修改一个Java文件后想看到效果不得不重启应用，这导致大量时间花费，我们希望不重启应用的情况下，程序可以自动部署（热部署）。有以下四种情况，如何能实现热部署。

- 1、模板引擎

  在Spring Boot中开发情况下禁用模板引擎的cache

  页面模板改变ctrl+F9可以重新编译当前页面并生效

- 2、Spring Loaded

  Spring官方提供的热部署程序，实现修改类文件的热部署

  下载Spring Loaded（项目地址https://github.com/spring-projects/spring-loaded ）

  添加运行时参数；

  -javaagent:C:/springloaded-1.2.5.RELEASE.jar –noverify

- 3、JRebel

收费的一个热部署软件

安装插件使用即可

- 4.Spring Boot Devtools（推荐）

  引入依赖

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

  IDEA使用ctrl+F9

  或做一些小调整

  Intellij IEDA和Eclipse不同，Eclipse设置了自动编译之后，修改类它会自动编译，而IDEA在非RUN或DEBUG情况下才会自动编译（前提是你已经设置了Auto-Compile）。

  设置自动编译（settings-compiler-make project automatically）

  ctrl+shift+alt+/（maintenance）

  勾选compiler.automake.allow.when.app.running

SpringBoot之监控管理

通过引入spring-boot-starter-actuator，可以使用Spring Boot为我们提供的准生产环境下的应用监控和管理功能。我们可以通过HTTP，JMX，SSH协议来进行操作，自动得到审计、健康及指标信息等

整合

1）、引入spring-boot-starter-actuator

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

2）、修改配置application.yml

```
management.security.enabled=false
spring.redis.host=118.24.44.169
info.app.id=hello
info.app.version=1.0.0
# endpoints.metrics.enabled=false
endpoints.shutdown.enabled=true
# endpoints.beans.id=mybean
# endpoints.beans.path=/bean
# endpoints.beans.enabled=false
#
# endpoints.dump.path=/du
# \u5173\u95ED\u6240\u6709\u7AEF\u70B9\u8BBF\u95E
```

```
# endpoints.enabled=false
# endpoints.beans.enabled=true
management.context-path=/manage
management.port=8181
```

3）、可进行shutdown（POST 提交，此端点默认关闭）

4）、通过http方式访问监控端点

端点名	描述
autoconfig	所有自动配置信息
auditevents	审计事件
beans	所有Bean的信息
configprops	所有配置属性
dump	线程状态信息
env	当前环境信息
health	应用健康状况
info	当前应用信息
metrics	应用的各项指标
mappings	应用@RequestMapping映射路径
shutdown	关闭当前应用（默认关闭）
trace	追踪信息（最新的http请求）

定制端点信息

- 定制端点一般通过endpoints+端点名+属性名来设置。
- 修改端点id（endpoints.beans.id=mybeans）
- 开启远程应用关闭功能（endpoints.shutdown.enabled=true）
- 关闭端点（endpoints.beans.enabled=false）
- 开启所需端点
  - endpoints.enabled=false
  - endpoints.beans.enabled=true
- 定制端点访问根路径
  - management.context-path=/manage
- 关闭http端点
  - management.port=-1

自定义HealthIndicator

1）、编写一个指示器，实现HealthIndicator

2）、指示器名字必须写 xxxHealthIndicator

3）、加入容器

```
@Component
public class MyAppHealthIndicator implatements HealthIndicator{
    @Override
    public Health health(){
        //自定义检查方法
        return Health.down().withDetail("").build;
    }
}
```