**1、系统架构的演变：**

1）**集中式架构**：代码耦合高，开发维护困难， 无法针对不同模块进行针对性优化，单点容错率低，并发能力差。

2）**垂直拆分**：系统间相互独立，会有很多重复开发工作，影响开发效率。

3）**分布式服务**：系统间耦合度变高，调用关系错综复杂，难以维护。

4）**服务治理（SOA）**：服务间会有依赖关系，一旦某个环节出错会影响较大，服务关系复杂，运维、测试部署困难。

5）**微服务特点**：自治（服务独立）、单一职责（一个服务对应一个业务能力）、团队独立、技术独立（使用rest接口各服务间技术不干涉）、前后端分离（采用前后端分离开发，提供统一Rest接口，后端不用再为PC、移动段开发不同接口）、数据库分离、部署独立（每个服务都是独立的组件，可复用，可替换，降低耦合，易维护）。

**2、远程调用：**

1）**RPC**：Remote Produce Call远程过程调用，类似的还有RMI。自定义数据格式，**基于原生TCP（三次握手）通信，速度快，效率高**。**早期的webservice，现在热门的dubbo，都是RPC的典型**。

**2）Http**：http其实是一种网络传输协议，**基于TCP**，规定了数据传输的格式。**现在客户端浏览器与服务端通信基本都是采用Http协议。也可以用来进行远程服务调用。缺点是消息封装臃肿。**  **现在热门的Rest风格，就可以通过http协议来实现。**

RPC规定调用方与服务方使用技术要一致，而Http则没有该要求。

**优点：RPC方式更加透明，对用户更方便。Http方式更灵活，没有规定API和语言，跨语言、跨平台  
缺点：RPC方式需要在API层面进行封装，限制了开发的语言环境。**

**3、Http远程调用详细。**

1. 流行的的http客户端工具：HttpClient（Apache公司的产品，是Http Components下的一个组件）、 OKHttp、URLConnection等。
2. HttpClient示例：

**//发起get请求：**  
    @Test  
    public void testGet() throws IOException {  
        HttpGet request = new HttpGet("http://www.baidu.com");  
        String response = this.httpClient.execute(request, new BasicResponseHandler());  
        System.out.println(response);  
    }

**//发起Post请求：**

@Test  
public void testPost() throws IOException {  
    HttpPost request = new HttpPost("http://www.oschina.net/");  
    request.setHeader("User-Agent",  
                      "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36");  
    String response = this.httpClient.execute(request, new BasicResponseHandler());  
    System.out.println(response);  
}

**得到的是一个json字符串需要反序列化为需要的对象：**JacksonJson工具是SpringMVC内置的json处理工具，其中有一个ObjectMapper类，可以方便的实现对json的处理。

**//对象转json**

  private ObjectMapper mapper = new ObjectMapper();  
     @Test  
     public void testJson() throws JsonProcessingException {  
        User user = new User();  
        user.setId(8L);  
        user.setAge(21);  
        user.setName("柳岩");  
        user.setUserName("liuyan");  
        // 序列化  
        String json = mapper.writeValueAsString(user);  
        System.out.println("json = " + json);  
     }

**//json转普通对象**

private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
    User user = new User();  
    user.setId(8L);  
    user.setAge(21);  
    user.setName("柳岩");  
    user.setUserName("liuyan");  
    // 序列化  
    String json = mapper.writeValueAsString(user);  
  
    // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
    User result = mapper.readValue(json, User.class);  
    System.out.println("result = " + result);  
}

**//json转集合**

private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
    User user = new User();  
    user.setId(8L);  
    user.setAge(21);  
    user.setName("柳岩");  
    user.setUserName("liuyan");  
  
    // 序列化,得到对象集合的json字符串  
    String json = mapper.writeValueAsString(Arrays.asList(user, user));  
  
    // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
    List<User> users = mapper.readValue(json, mapper.getTypeFactory().constructCollectionType(List.class, User.class));  
    for (User u : users) {  
        System.out.println("u = " + u);  
    }  
}

**//json转任意复杂类型**

private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
    User user = new User();  
    user.setId(8L);  
    user.setAge(21);  
    user.setName("柳岩");  
    user.setUserName("liuyan");  
  
    // 序列化,得到对象集合的json字符串  
    String json = mapper.writeValueAsString(Arrays.asList(user, user));  
  
    // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
    List<User> users = mapper.readValue(json, new TypeReference<List<User>>(){});  
    for (User u : users) {  
        System.out.println("u = " + u);  
    }  
}

3)Spring的**RestTemplate**模板工具类对基于Http的客户端进行了封装，并且实现了对象与json的序列化和反序列化,没有限定Http的客户端类型，而是进行了抽象，目前常用的3种都有支持：HttpClient、 OkHttp、J**DK原生的URLConnection（默认的）**。**见项目cousumer1示例。**

**4、Eureka：**服务注册中心（可以是一个集群），实现了服务的自动注册、发现、状态监控。见项目eureka-server示例。  
-- 提供者：启动后向Eureka注册自己信息（地址，提供什么服务），提供服务的应用，可以是SpringBoot应用，也可以是其 它任意技术实现，只要对外提供的是Rest风格服务即可。见项目user-service示例  
-- 消费者：消费应用从注册中心获取服务列表，得知每个服务方的信息，知道去哪里调用服务方。见项目cousumer1示例。

-- 心跳(续约)：提供者定期通过http方式向Eureka刷新自己的状态。

**5、负载均衡Robbin**：见项目cousumer1示例。

1)Eureka中已经集成了Ribbon，无需引入新的依赖。直接在RestTemplate的配置方法上添加`@LoadBalanced`注解。

2)修改调用方式，不再手动获取ip和端口，而是直接通过服务名称调用。

#Ribbon默认的负载均衡策略是简单的轮询,提供了**修改负载均衡规则**的配置入口，

格式是：{服务名称}.ribbon.NFLoadBalancerRuleClassName，值就是IRule的实现类

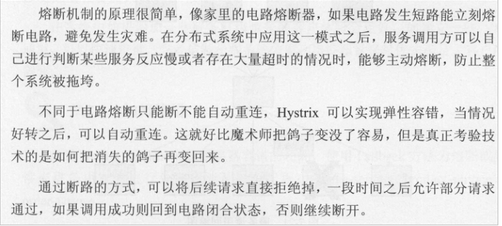
user-service. Ribbon. NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

#引入spring-retry依赖  
<dependency>  
    <groupId>org.springframework.retry</groupId>  
    <artifactId>spring-retry</artifactId>  
</dependency>

#修改配置

spring.cloud.loadbalancer.retry.enabled.true # 开启Spring Cloud的重试功能  
 user-service.ribbon.ConnectTimeout: 250 # Ribbon的连接超时时间  
user-service.ribbon.ReadTimeout: 1000 # Ribbon的数据读取超时时间  
user-service.ribbon.OkToRetryOnAllOperations: true # 是否对所有操作都进行重试  
user-service.ribbon.MaxAutoRetriesNextServer: 1 # 切换实例的重试次数  
user-service.ribbon.MaxAutoRetries: 1 # 对当前实例的重试次数

**6、配置Hystix熔断**：见项目cousumer2示例。



#引入依赖

<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

#改造user-consumer，添加一个用来访问的user服务的DAO，并且声明一个失败时的回滚处理函数：  
@Component  
public class UserDao {  
  
    @Autowired  
    private RestTemplate restTemplate;  
  
    private static final Logger logger = LoggerFactory.getLogger(UserDao.class);  
  
    **@HystrixCommand(fallbackMethod = "queryUserByIdFallback")**  
    public User queryUserById(Long id){  
        long begin = System.currentTimeMillis();  
        String url = "http://user-service/user/" + id;  
        User user = this.restTemplate.getForObject(url, User.class);  
        long end = System.currentTimeMillis();  
        // 记录访问用时：  
        logger.info("访问用时：{}", end - begin);  
        return user;  
    }  
  
    public User queryUserByIdFallback(Long id){  
        User user = new User();  
        user.setId(id);  
        user.setName("用户信息查询出现异常！");  
        return user;  
    }  
}

#在原来的业务逻辑中调用这个DAO：  
@Service  
public class UserService {  
    @Autowired  
    private UserDao userDao;  
  
    public List<User> queryUserByIds(List<Long> ids) {  
        List<User> users = new ArrayList<>();  
        ids.forEach(id -> {  
            // 我们测试多次查询，  
            users.add(this.userDao.queryUserById(id));  
        });  
        return users;  
    }  
}

#测试：改造服务提供者，随机休眠一段时间，以触发熔断：  
@Service  
public class UserService {  
  
    @Autowired  
    private UserMapper userMapper;  
  
    public User queryById(Long id) throws InterruptedException {  
        // 为了演示超时现象，我们在这里然线程休眠,时间随机 0~2000毫秒  
        Thread.sleep(new Random().nextInt(2000));  
        return this.userMapper.selectByPrimaryKey(id);  
    }  
}

实现熔断后重试机制似乎没有生效是因为R**ibbon超时时间设置的是1000ms，而Hystix的超时时间默认也是1000ms**，因此重试机制没有被触发，而是先触发了熔断。所以**Ribbon的超时时间一定要小于Hystix的超时时间**。

#设置Hystrix超时时间

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:6000ms

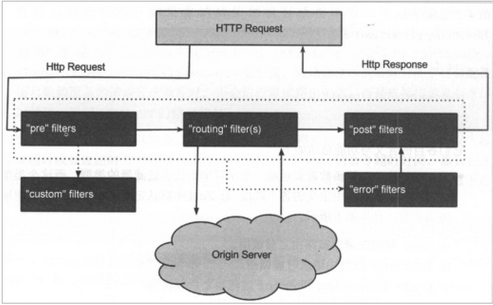
**7、Feign:**可以把Rest的请求进行隐藏，伪装成类似SpringMVC的Controller一样。你不用再自己拼接url，拼接参数等等操作，一切都交给Feign去做。封装了ribbon负载均衡和Hystix断路器。见项目comsumer1示例。

**8、Zuul网关:**服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了`权限控制`等功能。Spring Cloud Netflix中的Zuul为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。见项目zuul-demo示例。

ZuulFilter是过滤器的顶级父类。在这里我们看一下其中定义的4个最重要的方法：  
public abstract ZuulFilter implements IZuulFilter{  
    abstract public String filterType();  
    abstract public int filterOrder();  
    boolean shouldFilter(); // 来自IZuulFilter  
    Object run() throws ZuulException; // IZuulFilter  
}

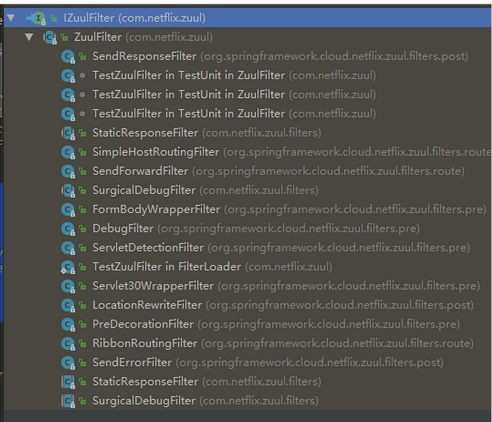
--shouldFilter：返回一个`Boolean`值，判断该过滤器是否需要执行。返回true执行，返回false不执行。  
--run：过滤器的具体业务逻辑。  
--filterType：返回字符串，代表过滤器的类型。包含以下4种：  
-- pre：请求在被路由之前执行  
-- routing：在路由请求时调用  
-- post：在routing和errror过滤器之后调用  
-- error：处理请求时发生错误调用  
--filterOrder：通过返回的int值来定义过滤器的执行顺序，数字越小优先级越高。

#过滤器执行生命周期



 正常流程：  
 --请求到达首先会经过pre类型过滤器，而后到达routing类型，进行路由，请求就到达真正的服务提供者，执行请求，返回结果后，会到达post过滤器。而后返回响应。  
异常流程：  
 --整个过程中，pre或者routing过滤器出现异常，都会直接进入error过滤器，再error处理完毕后，会将请求交给POST过滤器，最后返回给用户。  
 --如果是error过滤器自己出现异常，最终也会进入POST过滤器，而后返回。  
 --如果是POST过滤器出现异常，会跳转到error过滤器，但是与pre和routing不同的时，请求不会再到达POST过滤器

#所有内置过滤器列表:



使用场景  
-- 请求鉴权：一般放在pre类型，如果发现没有访问权限，直接就拦截了  
-- 异常处理：一般会在error类型和post类型过滤器中结合来处理。  
-- 服务调用时长统计：pre和post结合使用。

Zuul中默认就已经集成了Ribbon负载均衡和Hystix熔断机制。但是所有的超时策略都是走的默认值，比如熔断超时时间只有1S，很容易就触发了。因此建议我们手动进行配置：  
zuul:  
  retryable: true  
ribbon:  
  ConnectTimeout: 250 # 连接超时时间(ms)  
  ReadTimeout: 2000 # 通信超时时间(ms)  
  OkToRetryOnAllOperations: true # 是否对所有操作重试  
  MaxAutoRetriesNextServer: 2 # 同一服务不同实例的重试次数  
  MaxAutoRetries: 1 # 同一实例的重试次数  
hystrix.command. Default.execution. Isolation.thread.timeoutInMillisecond: 6000 # 熔断超时时长：6000ms