最近整理了一下自己所掌握的技术栈，发现自己所掌握的技术实在是太少了，无非就是使用几个简单的技术，搭建一个项目环境，然后进行编写代码。虽然现在自己创建的项目技术选型都是固定的那几种技术，如springboot、shiro、vue等，但是自己所能够做的东西都是比较简单，基本上都是增删改查的操作，对这些技术都没有太过深入的研究，只会比较基础的使用。在各大网站上搜了下比较出名的企业面试题，发现那些问题都好难啊，好大一部分都回答不出来，有的是考基础，有的则是考对流行技术的使用。

所以为了增加点自己的知识储备，打算在网络上找些视频教程继续学习一下内容。这次我找的是spring cloud的视频教程。现在互联网发展迅猛，一个网站系统每日的工作量是巨大的，将项目分开部署、搭建集群是现在最通用也是最有效降低服务器负担的方法。spring cloud正是为了微服务所诞生的技术，之前在github搜索项目时，也发现了好多项目都是选用了这项技术，所以这也是我选这门技术学习的原因之一。

软件工程专业，想要技术有所提升，多敲代码是难免的一项步骤。视频的第一节课，老师对这次视频教程进行简单概述，要求我们在看同时，也要跟着视频敲代码，巩固学习。并且老师在视频的最后说了一句话“不在能知，乃在能行。知行合一，方能进步”。

学习一项技能之前，必须要先理解其概念。理论掌握好了，学习与使用便会更加顺手。老师在开头先是介绍了微服务架构的理念。分久必合合久必分，微服务架构提倡将单一应用程序划分成一组小的服务，并且服务与服务之间互相协调、互相配合。

之前认为spring cloud像是阿里巴巴的dubbo框架一样，是一项RPC的技术框架。但是经过老师的介绍，发现原来之前对spring cloud的了解大相径庭。Spring cloud是一系列框架的有序集合，它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。

Spring cloud它将各大产商较为成熟的框架结合起来，其中包含了大概二十种左右的技术。虽然里面已经大部分技术已经过时并且停止更新，但是还是需要对其有的技术进行了解，如果在之后如果接触到，能够对其有一个大致的印象。虽然一些技术已经被抛弃，但是他们的设计理念大部分得到保留，之后新出现的技术，可能遵循的也是一样的架构。比如RPC的设计，遵守的都是消费者、服务提供者和注册中心之间的相互通信。像是冯诺依曼的计算机体系，他是早在一九四五年提出，但是到现在计算机依然遵守的是如此的设计架构。所以学习知识是不会无用，总是会有其发挥作用的地方。

今天学习的主要是服务的注册与发现，老师在这个系列中会介绍eureka、zookepper、consul和nacos的使用。虽然需要学习的技术有好几种，并且在一个系统的开发中只会选取一种进行使用，但是对于这些技术，必须都要有所掌握与了解，权衡各项技术中的优点与缺点，选出最适合当前项目的技术。并且掌握的技术越多，在今后的工作中，对公司所选择的技术能够有所了解与使用，有利于今后的开发工作。

老师先是不使用框架进行微服务的调用以此进行作对比，所使用的是RestTemplate类中的方法，直接填写微服务的地址进行调用。使用此方法进行调用，开发时必须知道所调用服务的地址，开发较为复杂。

约定大于配置，配置大于编码。在对项目导入一项技术时，对于环境的搭建是至关重要的。环境搭建错误，后面的编码都将不起作用。老师在导入eureka配置时，打开了思维导图文件，其中记录了导入框架时的每一步操作。并且按步骤顺序罗列好，对于该环境的搭建有了十分直观的感受。

经过技术的迭代更新，eureka从之前导入的同一个jar包，变成服务端与客户端的jar包分开导入，对于两端有了更明显的划分。将eureka搭建完成后，创建了多个同类的服务提供者注册进eureka的服务端中，体验到消费者对于服务的调用更加的方便，但是对于单点故障的原因还没有得到很好的解决。

今天继续学习服务的注册与发现中的zookeeper和consul框架的使用。类似功能的框架感觉大部分都是换汤不换药。底层原理都是相似的。Eureka、zookeeper和consul他们都是充当服务注册中心的角色，使用步骤基本一致，服务提供者注册，消费者获取使用。

在搭建成功eureka之后，接下来是搭建zookeeper的环境，在两者切换时，只需要替换一个专属jar包、修改application.yml文件和更改启动类的一个注解便能够生效使用。Consul也是与前两者类似，业务代码不需要变更，只需要更改配置便能够进行替换。

最后学习了这三个框架的不同点。老师先是介绍了CAP的核心理论，一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求。C表示一致性，A表示可用性，P表示分区容错性。Eureka、zookeeper和consul他们根据CAP架构可以分成CP、与AP。Zookeeper和consul划分为CP，使用他们作为注册中心时，一旦服务提供方不可用时，就直接将其剔除，不会对其进行保留。Eureka划分到AP，因为eureka默认开启自我保护机制，如果服务提供方出错时，eureka会对其进行保留一段时间，期待之后能够继续使用。

不能同时满足CAP的原因是，可能机器之间难免会出现不可能避免的故障，如网络异常等情况，系统A的数据不能同步到系统B，导致B的数据是异常的。如果需要保证B系统可用就不能保证数据一致性，剔除B系统便会失去可用性的特点。

学习完服务的注册与发现后，接下来便是学习负载均衡工具的使用。这次所学习的工具是Spring cloud ribbon，ribbon是基于Netflix Ribbon实现的一套客户端负载均衡的工具。简单地说就是，Ribbon主要功能是提供客户端的软件负载均衡算法和服务调用。Ribbon客户端组件提供了一系列完善的配置如连接超时，重试等。项目在使用他时，只需要在application.yml中编写相应的配置便能够生效。Ribbon会找出服务提供方的所有机器，然后自动的执行某项规则，如随机连接、简单轮询等自带的七种规则去链接这些机器。同样是具有负载均衡功能的工具是nginx，ribbon与nginx的最大区别是，ribbon是在客户端进行负载均衡，而nginx是在服务端所进行的。Nginx会拦截所有相应的请求，然后转发到对应的机器上，客户端只需要访问nginx就行。而ribbon是在所有的服务方地址缓存进自己的jvm虚拟机，然后通过自己的计算，自行访问相应的机器。

工具不仅仅只会是去用，还需要去了解他的原理，了解它的实现。老师打开ribbon的源码，让我们去了解Ribbon负载规则算法的实现。老师让我们先去看轮询算法的实现，基本原理便是，客户端去注册中心获取到所有可用的服务提供方地址，然后存进一个集合中进行保存，之后根据请求，每次请求访问次数就加1，然后集合总数除以访问数求余，最后根据结果去集合中get(index)获取目标主机。其中在RoundRobinRule类中，记录访问次数的变量时，它使用了AtomicInteger原子类，并且使用自旋锁的方式，确保了每次访问该变量的线程安全。

今天学习的是服务的调用OpenFeign的使用。Openfeign是spring cloud在feign的基础上支持了spring mvc的注解，使之使用起来更加的方便。Feign旨在使编写Java Http客户端变得更加容易。在之前的学习中都是使用Ribbon+restTenplate进行使用，利用restTenplate对http请求进行封装处理，形成了一套模板化的调用方法。但是在实际开发环境中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，在Feign在此基础上做了进一步封装，由它来帮助我们定义和实现依赖服务接口的定义，在feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它，即可完成对服务提供方的接口绑定，简化了使用spring cloud ribbon时，自动封装服务调用客户端的开发量。

Feign在定义调用接口时，与Mybatis定义的Mapper接口十分类似，都是编写接口方法，填入相应的注解，框架便会自己去调用内部的方法。Feign是在服务调用方进行定义使用，在使用时，需要在启动类上使用EnableFeignClients注解，使Feign能够在项目中进行生效。并且在定义Feign的接口时，方法与调用方controller层的接口方法完全一致。使开发时，提高了面向对象编程的理念。Feign还提供了超时控制与日志记录的功能，更加完善了feign在使用时的调用逻辑，避免长时间等待，影响程序的正常运行。

今天对服务降级、熔断的实现技术进行学习，这次所学习的是Hystrix框架。老师先是讲解了hystrix框架提出的原因，并且说明为什么需要使用它。现在的系统基本上都是基于分布式进行部署的，复杂的分布式体系结构中的应用程序之间存在着很强的依赖关系，每个依赖关系在使用的过程中不可避免的会出现失败的调用，这时候就需要对这些失败的请求进行处理，保证系统的正常运行，防止出现服务雪崩的情况。

老师介绍可能会出现服务雪崩的情况。当服务A调用服务B，服务B又调用其它的服务，这就是造成服务的扇出。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，这就是所谓的“雪崩效应”。这些应用程序可能导致服务之间的延迟增加，致使线程和其它的系统资源紧张，导致整个系统发生更多的级联故障，让整个分布式系统瘫痪。

接下来在自己的电脑上模拟了一次因为调用的延迟时间正常，导致其它接口的方法也随之增加的案例。先是编写controller层的两个接口，接口A正常运行，接口B在执行时停顿一段时间，模拟网络延迟。在正常情况下请求接口A与B时两个互不干扰，运行正常。但是经过jmeter进行压力测试，使用多线程重复调用接口B，使调用资源占满。接下来再访问接口A时，访问出现了明显的卡顿情况。经过此次试验，更加了解了处理失败请求的重要性与服务熔断框架出现的必然原因。

Hystrix的继续学习。昨天学习了Hystrix出现的原因与解决的问题后，今天继续学习hystrix的概念。Hystrix的概念是用于处理分布式系统的延迟和容错的开源库，在分布式环境中，许多服务依赖项中的一些必然会失败。Hystrix是一个库，通过添加延迟容忍和容错逻辑，帮助开发人员控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点、停止级联失败和提供回退选项来实现这一点，所有这些都可以提高系统的整体弹性。

Hystrix提供服务降级方法与断路器的设置。可能发生服务降级的情况，一是超时调用，二是服务提供方本身错误，三是服务方宕机。断路器本身是一个开关装置，当某个服务单元发生故障之后，通过断路器的故障监控，类似于生活中的保险丝，像调用方返回一个符合预期的、可处理的备选响应，而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要的占用，从而避免故障在分布式系统中的蔓延，乃至雪崩。

断路器打开的必要的避免条件，在一个时间窗口内，请求达到一定的次数，并且错误率达到阈值，此时断路器便会打开。在经过一段时间后，Hystrix会再次尝试请求服务错误方，如果恢复正常，断路器变成半打开状态，直至关闭。

Hystrix环境的搭建。因为都是与spring cloud有了很好的集成，所以在添加Hystrix环境时，只需要添加一个spring-cloud-starter-netflix-hystrix的jar就行，接下来便是要修改配置文件，与添加业务代码逻辑。Hystrix可以添加在服务调用方或者是被调用方。不管是添加在哪一方，添加步骤都是一样的。在要使用Hystrix的微服务的启动类上，必须添加EnableCircuiBreaker注解，以此开启该框架的功能。在学习了spring cloud之后，发现在使用各项技术框架时，大部分都是使用注解的方式使用其功能，使开发时更加方便了。配置降级方法和熔断就主要的两个注解便是HystrixCommand和HystrixProperty。其中HystrixProperty需要配置感觉是最难记的，类似设置超时时间是使用execution.isolation.thread.timeoutInMilliseconds，这个都是需要自己手写。还好在HystrixCommandProperties类中有对这个注解配置的所有归纳，在之后想要配置时，可以打开这个类进行查找。

接下来便是对环境进行测试，先是测试调用时间超时的情况，在测试接口设置程序了睡眠时间，浏览器正常显示了超时后降级方法的处理。但是在使用时，发现降级方法的参数必须与接口的参数一致，否则一直都是提示该降级方法没有定义，觉得这个使用有点麻烦，比如有时候想要在超时调用时，统一调用一个降级方法，避免代码的膨胀。在尝试了多次后也没有寻找到解决办法，就看之后再查找查找资料，寻找下解决方案。

今天学习服务网关spring cloud gateway的使用。理论、实操和小总结三步骤，以此来学习能够更好的理解技术的使用。Spring cloud gateway它旨在为微服务架构提供一种简单有效的统一的API路由管理方式。Spring cloud gateway作为spring cloud生态系统中的网关是基于WebFlux框架实现的，而webflux框架底层则是使用了高性能的Reactor模式通信框架Netty。它的目标提供统一的路由方式且基于Filter链的方式提供了网关的基本功能，例如：安全，监控和限流等。

接下来老师讲解在servlet3.1架构的缺点，以此来阐明异步非阻塞提高性能的重要性。Servlet是一个简单的网络IO模型，当请求进入时，它就会为其绑定一个线程，在并发量不高的情况下，这种模型是可以满足基本需求的。但是一旦高并发时，线程数量就会上涨，而且线程资源代价是昂贵的，这会严重影响请求的处理时间。

开始配置spring cloud gateway环境，因为gateway中使用webflux，这与spring-boot-starter-web中的jar冲突，所以需要删除该jar包。接下来便是修改配置文件，在这中间遇到了一个因为大意而出现的bug。在配置时，不小心将uri写成了url，在测试时，浏览器一直提示404 not found的错误。随后还去网上找了下url与uri的区别，加深对这两个使用的印象。Spring cloud gateway有两种配置方式，除了在配置文件编写外，也可以在springboot的配置类中进行编写，加上Bean注解并且放回RouteLocator类便可以实现，书写方法都是与配置文件书写类似。

今天学习Spring cloud中配置中心config的使用。微服务的出现意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务，由于每个服务都有需要配置的必要信息才能运行，所以一套集中式的，动态的配置管理设施是必不可少的，配置中心的出现是使修改一处，处处生效。Spring cloud config分为服务端和客户端两部分。服务端被称为分布式配置中心，用来连接配置服务器并为客户端提供获取配置信息的接口。客户端则是通过服务端来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring cloud config配置服务器默认使用Git来存储信息。

Config的配置文件分成两个，application和bootstrap，前一种是用户级的资源配置项，后一种是系统级的，优先级相比较前一种来得更高。在初始化时，bootstrap负责从外部加载配置信息，默认情况下它不会被本地配置覆盖。

接下来对spring cloud config环境的搭建，首先需要创建服务端，设置该服务的配置，指明所使用的配置服务器和仓库地址，并在启动类加上EnableConfigServer注解，完成配置。客户端的配置与服务端相似，导入相应的jar包后，创建bootstrap文件，指明所依赖的配置中心的地址，便能够生效。单独使用spring cloud config有一个弊端，在配置中心配置的信息不能够自动的刷新数据，需要对每一个客户端访问一个接口，才能够生效其修改的配置，如果微服务的数量较多时，修改的工作量也会比较大，需要逐一配置生效。

继续对Spring cloud进行学习。Spring cloud存在消息总线的框架bus。在微服务架构的系统中，通常会使用轻量级的消息代理来构建一个公用的消息主题，并让系统中所有微服务实例都连接上来。由于该主题中产生的消息会被所有实例监听和消费，所以称他为消息总线。在总线上的各个实例，都可以方便地广播一些需要让其它连接在该主题上的实例都知道的消息。Spring cloud bus能够解决之前学习的config配置中心不能自动更新存在的问题。配置客户端都监听MQ中同一个topic。当一个服务刷新数据的时候，他会把这个信息放入到topic中，这样其它监听同一个topic的服务就能够收到通知，然后去更新自身的配置。

老师讲解完概念后，让我们思考消息总线需要配置在哪个服务器上，有放在一个configclient实例上和单独抽取出一个服务。因此来加深对微服务的理解。最后答案是单独抽取出一个服务。原因有三点，微服务的本身职责是单一的，因为微服务是业务模块，不应该承担另外的配置刷新的职责。防止破坏微服务各节点的对等性。避免提高项目的局限性，例如，微服务在迁移时，他的网咯地址发生变化，此时如果想要做到自动刷新，那就回增加更多的迁移操作。

在配置好spring cloud bus的环境后，在修改配置服务中心的数据后，由之前通知每一个实例进行更新后，变成现在只需要通知spring cloud bus的实例进行更新，其他的配置实例便会受到通知，进行更新。