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。

今天开始对Spring cloud stream初步学习。Spring cloud stream是搭配消息中间件进行使用。在我们实际的开发过程中，服务与服务之间的调用经常会使用到消息中间件，以此来解决系统异步的操作，提高运行效率。但消息中间件多种多样，学习需要成本，并且直接使用中间件的话，系统的耦合度便会增高，如果要进行替换，耗费的成本将会比较大。Spring cloud stream整合了消息中间件，来降低系统和中间件的耦合度，目前仅支持kafka与RabbitMQ两种消息中间件。

应用程序通过inputs或者outputs来与Spring cloud stream中binder 交互，通过我们配置来binding，而Spring cloud stream的binder负责与消息中间件交互。所以，我们只需要搞清楚如何与Spring Cloud Stream交互就可以方便使用消息驱动的方式。相当于Stream对消息中间件进行了一层封装，可以做到代码层面对中间件无感知，使得微服务的开发高度解耦，服务可以关注更多自己的业务逻辑。

Stream常用的四大注解。Input: 标识输入通道，通过该输入通道接收到的消息进入应用程序; Output: 标识输出通道，发布的消息将通过该通道离开应用程序; StreamListener: 监听队列，用于消费者的队列的消息接收; EnableBinding: 指信道channel和exchange绑定在一起。为了防止消费者对消息进行重复消费，导致数据出现错误。Stream还提供了分组的功能。每一组中存在N个成员，成员之间对消息消费属于竞争关系，如果一个消费了，其它成员就不会进一步执行。

学习Spring Cloud Sleuth概念与环境搭建。Sleuth出现的原因是微服务架构是一个分布式架构，一个分布式系统往往有许多个服务存在。由于服务数量众多，业务的复杂性，如果出现了错误和异常，开发人员如果要去定位的话，就很艰难。主要体现在，一个请求可能需要调用很多个服务，而内部服务的调用复杂性，决定了问题难以定位。所以微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎样的，从而达到每个请求的步骤清晰可见，如果系统中出现问题，就能够快速定位。

老师在讲解是举了个例子帮助理解，在微服务系统中，一个来自用户的请求，请求先达到前端A，然后通过远程调用，达到系统的中间件B、C，最后达到后端服务D、E，后端经过一系列的业务逻辑计算最后将数据返回给用户。对于这样一个请求，经历了这么多个服务，怎么样将它的请求过程的数据记录下来呢？这就需要用到服务链路追踪。

Span是基本工作单元，发送一个远程调度任务 就会产生一个Span；Trace是一系列Span组成的一个树状结构。Sleuth是懒加载的机制，当用户发起第一次请求后，才会对链路进行加载记录，避免一次性全部加载，导致启动缓慢。Sleuth搭配Zipkin进行使用，将分布式系统中的链路信息在网页上更加直观的展示给开发人员。

接下来对spring cloud alibaba的组件进行学习，最先学习的是nacos框架。Nacos包含了多个方向的应用，有服务注册与发现、配置中心和服务的降级熔断。Nacos支持AP和CP模式的切换，可以根据实际情况进行更换。一般来说，如果不需要存储服务级别的信息且服务实例是通过nacos-client注册，并能够保持心跳上报，那么久可以选择AP模式。当前主流的服务Spring cloud和Dubbo服务，都是用于AP模式，AP模式为了服务的可能性而减弱了一致性，因此AP模式下只支持注册临时实例。如果需要在服务级别编辑或者存储配置信息，那么CP是必须的，K8S服务和DNS服务则适用于CP模式。CP模式下则支持注册持久化实例，此时则是以集群运行模式，该模式下注册实例之前必须先注册服务，如果服务不存在，则会返回错误。

技术好像大部分都是换汤不换药，虽然前面的技术有时候会被之后的技术所取代，但是一开始的中心思想，大部分是会被借用的。在搭建nacos的环境时，与之前的consul类似，都是文件下载成功使用命令单独的启动，在springboot中进行相应的配置便能够将服务注册进nacos中。各个技术之间的耦合度很低，在只是把服务注册中心切换后，调用可以都不需要进行更改，依然能够使用RestTemplate或者Openfeign进行调用。对当中的执行流程感知很低，可以专注与编写业务代码。Nacos的服务默认是临时态，在服务停止心跳后，便将服务在nacos进行清除。

对Nacos进行进一步的学习配置。Nacos安装好后，打开服务端的页面，里面有着十分全面的管理信息。除了服务管理设置服务注册外，另一个最主要的便是配置管理。Nacos也可以当做配置中心以此来替代spring cloud config。接下来就是对nacos的配置中心进行搭建测试。

Nacos的配置中心与spring cloud config的配置大体相似。spring cloud config需要结合Git或者是其它的代码托管平台进行配置信息的存储，操作相比较于nacos较为复杂。Nacos可以直接在服务端进行配置文件的编写和存储。Nacos默认存储方式是将数据保存在本地磁盘中进行读取。使用Nacos配置中心，客户端与之前一致。都是需要创建bootstrap.yml文件，优先级比application.yml高。在每次启动项目时，都会去配置中心拉取配置信息，然后将信息进行写入。

Nacos相比较与spring cloud config还具有一个显著的优点，config需要结合spring cloud bus进行配置的动态刷新，nacos已经很好的解决了这方面的的短板。在nacos修改了相关的配置信息后，点击发布，客户端便会受到通知配置信息已经得到了修改，并且进行动态的刷新，这个需要在配置了nacos的注解下进行完成。为了进行隔离，防止不同的配置可能会进行混淆。Nacos提供了DataId、Group分组和Namespace命名空间的三种概念。这三种概念相当于分别对应Java项目的类名、包名、项目名，这对文件进行了很好的隔离。Group默认为DEFAULT\_GROUP，Namespace默认为public，客户端可以在配置文件进行指定，选取与之相对应的配置文件。

一个分布式微服务架构的系统，总是避免不了集群的搭建。使用Nacos作为服务注册中心也是一样的，生产环境中不可能只有一台在运行的。如果使用单机版，那么Nacos发生了一些不可避免的错误时，，所有的服务链路都将中断，整个分布式系统陷入瘫痪。为了尽可能避免这种情况的出现，所以需要对Nacos进行集群的搭建。

老师在讲解的过程中是使用Linux环境进行搭建的。首先第一步是需要解决Nacos的存储问题，因为默认Nacos是存储在本地的文件中，使用这种存储方式，难以保证集群中所有信息的一致性。所以需要将Nacos的数据放进Mysql数据库中进行存储，所有的Nacos实例都从同一个地方的数据库进行获取，就能保证数据的一致。Nacos的配置文件夹中有Mysql数据库的脚本文件，需要将脚本文件执行，创建Nacos所能够识别的数据库表。其中修改Application.properties文件，改变Nacos的存储方式，设置数据库的地址、用户名和密码。接下来是修改cluster.conf文件，该文件时设置集群中每个机器的地址。文件设置较为简单，只需要将ip地址和端口号写入。

因为在测试时，想要在同一台机器运行多个nacos实例。所以需要修改nacos的启动端口，进行分开的运行。需要修改到的便是bin目录下的startup文件。老师所使用的是linux系统，所以为了能够思考，我便在windows下进行修改。因为已经有写好的脚本，我只需在原有的基础上进行修改，所以完成的也较为简单。最终在启动命令上添加了自己的启动参数，可以动态设置了Nacos的启动端口，最后也是测试通过，搭建成功。

Spring cloud alibaba在spring cloud社区提供了许多强大的技术，这其中就包括了设置服务熔断的sentinel框架。与之前学习的hystrix不同的是，sentinel把大部分的配置转移到了服务端进行配置，减少了项目中编写的代码，降低了代码的耦合度。在从官网下载了Sentinel控制台的启动文件后，便能够直接启动运行。Sentinel默认的启动端口是8080，该端口容易被占用，在启动的时候可以添加命令-Dserver.port来指定启动的端口。Sentinel下载下来的是一个jar文件，也可以用解压软件打开，修改其中的application.yaml进行配置的修改，但是该办法比较局限，修改起来相比较于前者较为麻烦。

Sentinel实现的是懒加载方式，只有当服务被运行一次过后，才会在控制台中进行记录，一开始打开控制台显示的只有一个首页信息。在请求了一次服务的controller层接口后，sentinel就显示出来相应的信息。里面包含了实时监控，查看当前请求的QPS，还有簇电链路、流控规则和降级规则等设置。

Sentinel可以对服务的流量进行控制。它是通过资源名来绑定流量控制的位置。资源名默认是请求的URL，同时也可以通过在controller层接口添加SentinelResource注解来添加资源名。流量控制主要是对QPS或者是系统的线程使用数进行控制。为了测试一下Sentinel的效果，显示对阈值类型设置成QPS，阈值设置成1，流控模式和流控效果分别设置成直接、快速失败。通过测试，当访问量过大时，正常弹出了降级方法的信息，Sentinel的初体验成功了。

今天继续对Sentinel的基础配置进行学习。昨天学习到流控规则其中的一个设置，Sentinel对流控有着多方面的配置。阈值类型包括QPS和线程数，流控模式其中包括直接、关联和链路，流控效果包括快速失败、Warm Up和排队等待。降下来是对这些概念进行理解，并且进行搭建测试设置后的效果。

QPS是指每秒中的请求数量。流控模式：直接是指当api达到限流条件时直接限流；关联是当关联的资源达到阈值时，就限流自己；链路是只记录指定链路上的流量，指定资源从入口资源进来的流量，如果达到阈值，就进行限流。流控效果：快速失败是直接失败，抛出异常；Warm Up是根据冷加载因子的值，默认是3，从阈值/冷加载因子，经过预热时长，才达到设置的QPS赋值。Warm Up当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。通过“冷启动”的方式，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上线，给冷系统一个预热的时间，避免冷系统被压垮。

Sentinel有着自己默认的降级方法，但是在实际开发中并不太实用。SentinelResouce注解提供了可以自己定义降级方法。其中实用blockHandler来指定流控的处理方法，fallback用来指定程序出现异常后的降级方法。为了避免增加代码耦合，Sentinel还可以将降级方法单独放在一个类中进行处理，但是方法必须指定为静态方法，使用blockHandlerClass或者fallbackClass来指定处理的类。

Sentinel的配置是十分全面的，今天对降级规则和热点规则进行配置。降级规则的策略其中包括平常响应时间、异常比例和异常数。需要触发Sentinel的熔断，需要1S内持续进入N个请求，Sentinel默认是5个，并且达到上述配置的信息，服务熔断才会打开。与Hystrix不同的是，Hystrix的熔断器有打开、关闭、半打开三种状态，在半打开的状态时，Hystrix会选择通过一些请求去判断服务是否恢复正常。而Sentinel只有打开和关闭两种状态，如果Sentinel的熔断器打开后，当达到设置的时间窗口后，熔断器就会处于关闭的状态。热点规则的配置是对请求的参数进行限制，可以对请求参数是否存在和其中的请求值进行设置，热点控制可以当做一个特殊的流量控制。

Sentinel默认是将配置信息写入内存中，如果在对Sentinel进行重启后，所有的数据就会消失和重新配置。这在生产中显然是不可能的。为了解决这个问题，Sentinel提供了对nacos和apolo等数据源的支持，可以将数据存储进这些数据源中，当Sentinel启动时，自动读取这些数据达到持久化的效果。本次学习的是使用nacos作为数据源。因为Sentinel默认提供的存储方式，只有从数据源取数据，不能将更新同步到数据源。所以需要将配置信息改变到Nacos进行编写。但是Nacos编写的配置信息是以json的方式编写，想比较与在Sentinel直接配置，变得稍微比较复杂。如果需要让Sentinel修改的信息同步到数据源，就要从github上下载他的源码，实现其特定的接口，对同步方法进行修改。

在使用到微服务的架构模式时，不可避免的会遇到分布式事务的问题。造成的原因是数据库会存在分库和分表的操作，来提升系统的执行效率。但微服务把单体应用被拆分成微服务应用，原来的模块被拆分成独立的小模块，他们可能分别使用不同的数据源。此时每个服务内部的数据一致由本地事务来保证，但是全局的数据一致性问题没法保证。

Spring cloud alibaba seata中间件很好的解决了这个问题。他采用一个中央事务控制器，记录一个事务中的所有操作，最后来判断事务需要全部回滚。其中有三个最主要的概念，TC（Transaction Coordinator）事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或者回滚。TM（Transaction Manage）控制全局事务的边界，负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。RM（Resource Manage）控制分支事务，负责分支注册、状态汇报，并接受事务协调器的指令，驱动分支（本地）事务的提交和回滚。

开始搭建seata环境，验证seata执行后的效果。因为seata1.0以上，文件发生了重要变化，本次使用的是seata0.8版本。因为要模拟数据库分库的操作，先是创建两个数据库和微服务项目。还要修改seata的配置文件，将数据存储进mysql数据库中。当环境全部搭建好时，只需在事务的开启入口添加@GlobalTransaction注解，便能够使全局事务生效。当在微服务的链路下层尝试抛出一个异常时，两个数据库的数据都没有得到修改。Seata全局事务的管理完成。

在昨天完成seata的搭建，看到它的执行效果后，今天便是学习seata的执行流程，加深对seata的印象。TM先开启分布式事务（TM向TC注册全局事务记录）；按业务场景，编排数据库、服务等事务内资源（RM向TC汇报资源准备状态）；TM结束分布式事务，事务一阶段结束（TM通知TC提交或者回滚分布式事务）；TC汇总事务信息，决定分布式事务时提交还是回滚；TC通知所有RM提交或者回滚资源，事务二阶段结束。

因为seata的工作机制，使得当需要发生事务回滚时，需要更多的处理时间。Seata AT模式的工作机制是：解析SQL，得到SQL的类型，表，条件等相关信息，并查询前镜像，根据解析到的条件信息，生成查询语句，定位数据。执行业务SQL，修改数据库信息。查询修改后的镜像，根据前镜像的结果，通过主键定位数据。插入回滚日志，把前后镜像数据以及业务SQL相关的信息组成一条回滚日志记录。在本地事务提交前，向TC注册分支。本地提交。将本地事务提交的结果上报给TC，TC再判断事务是否需要回滚。

Seata只是对数据库修改前后的镜像进行保存，本地事务都是照常提交到数据库，当在项目中打断时，发现整个流程还没有执行结束，数据库已经有了相应的数据。当执行出错事务需要回滚时，seata根据前镜像，再将数据库的数据还原回去，实现数据的一致性。而且seata在执行时，都需要将日志和镜像存储进数据库，在最后执行结束后，又将这些记录删除，这是降低系统执行时间的情况下，来保证数据库数据的一致性。

在生产的实际开发中，不可避免的运行会出现很多错误，在高并发下，程序经常会发生逻辑上的错误，并且这种错误还是不好排查。所以在今天开始学习Java在业务开发中的常见错误。

像是有时候服务器运行不稳定，很可能是代码上存在着逻辑漏洞，很多时候遇到OOM、死锁、超市问题在运维层面通过修改配置、重启、扩容等手段解决了，没有反推到开发层面去寻找原因。有时候问题会在特定的情况下发生，比如，缓存击穿、在多线程环境使用非线程安全的类，只有在多线程或高并发的情况下才会暴露问题。有些性能问题不会导致明显的bug，只会让程序运行缓慢、内存使用增加，但会在量变到质变的瞬间爆发。

在学习文章的末尾，看到了一个存在的常见错误。在毕业设计中刚好遇到了这个问题，之前输出日志，没有找到出现的问题。SImpleDateFormat是线程不安全的类，当把他设置成类的成员变量时，当多线程同时使用同一个SImpleDateFormat类进行时间转化时，可能会出现时间转化错误的问题。测试了一下SImpleDateFormat不安全的例子，·发生错误的效果，与毕业设计中的结果一致。搜索了下这个问题的解决方法，可以将SImpleDateFormat放进ThreadLocal容器中，每个线程拥有独立的实例，这样在多线程的环境下对时间的转化，就可以做到互不干扰，使转化出来的时间都是正常的。

在一些并发的场景下，通常类的成员变量会使用并发工具类库中定义的类，来保证数据的线程安全性。比如一般的都认为把HashMap改为ConcurrentHashMap，就可以完全的解决并发的问题。有时候可能也会使用无锁设计的CopyOnWriteArrayList来代替，提升系统的性能。但是这些说法一般来说都不太准确。

在JDK1.5后退出的ConcurrentHashMap，是一个高性能的线程安全的哈希表容器。“线程安全”特别容易被误解，ConcurrentHashMap只能保证原子性的读写操作是线程安全的。在之后使用了一个例子来验证这个问题的准确性。先创建一个含有900元素的Map，然后在检测当前Map中的数量距离1000还有多少个元素，然后将补充的元素存放进去，最后输出得到总数。在单线程的情况下，这个是没有问题的。但是在多线程的情况下并行去执行这个操作时，错误就出现了，最后输出的总数很容易大于1000。

造成这个原因是，在一个线程判断Map还需要添加100个元素后，往Map中添加元素，但是还没有添加结束的时候，在这个中途，其他线程进来了判断，发现Map可能还剩余0-100之间的元素个数需要添加，所以又往Map中存放元素，这就导致了数据添加过多的情况。PutAll这个函数方法是不能够保证数据操作的原子性，如果在putAll的过程中，其他线程进来判断集合中的个数，往往不能够得到最真实的结果。为了要解决这个问题，最简单的方法就是在判断个数和存放数据时添加锁，避免其它线程进来干扰，影响结果的正确性。

之前在使用ThreadLocal来解决问题，但是ThreadLocal没有使用好的话，也有可能导致信息错乱的错误。ThreadLocal适用于变量在线程间隔离，如果用户信息的获取比较昂贵（比如从数据库中查询用户信息），那么在ThreadLocal中缓存数据是比较合适的作答。但处理不好时，也有可能会出现信息错乱的bug。在使用ThreadLocal获取信息时，有时候获取到的是别人的。

会出现这个问题的原因是，有时候线程池会重用固定的几个线程，一旦线程重用，那么很可能首次从ThreadLocal获取的值是之前其他用户的请求遗留的值。这是，ThreadLocal中的信息就是其他用户的信息。

为了验证，使用SpringBoot创建一个web应用程序，使用ThreadLocal存放一个值，这个初始值的null。在代码的编写中，先从ThreadLocal获取第一次值，然后把外部传入的参数设置到ThreadLocal中，最后再获取一次值。为了明显的体现出效果，在web服务器的最大线程数设置1。接下来对程序进行测试，先传入参数1，看到控制台的输出都是正常的，第一次输出null，第二次输出1。而在第二次请求传入参数2时，控制台的信息就出现了问题，第一次出错的显示了1，第二次正常显示2。

因为线程的创建比较昂贵，所以Web服务器往往会使用线程池来处理请求，这就意味着线程会被重用。这时，使用类似ThreadLocal工具来存放一些数据时，需要特别注意在代码运行完后，显示地去清空设置的数据。

HashMap是一个线程不安全的类，在多线程的环境中不建议使用。在jdk1.8之前，HashMap在多线程的情况下，可能会在扩容的时候造成死循环或者造成数据的丢失的情况。

为了测试HashMap在多线程的环境下，可能会出现的问题，写了一段代码进行模拟。代码中创建多个线程，并行的对HashMap进行put操作，尽量多的设置循环的put的次数。在运行这段代码时，发现没有一会，电脑的cpu就被打满，并且程序无法继续往HashMap中put数据。有时候控制台会抛出数组越界的异常，而有时候则控制台没有任何信息。

当控制台没有信息时，通过jps和就stack命名查看情况，发现线程在进行着一个循环操作，从信息中看出了循环的位置，在HashMap的扩容函数transfer中。定位到该函数中，查看出现的原因，发生jdk1.8之前，HashMap使用的是头插法，也就是链表在扩容的时候会发生翻转。当一个线程resize完成，另一个线程未resize完成，连表会指向resize之后的链表，另一个线程的e.nxet()指向了前一个，导致了死循环。

在jdk1.8中对HashMap进行了优化，在发生hash碰撞，不再采用头插法的方式，而是直接插入链表尾部，因此不会出现环形链表的情况，但是在多线程的情况下仍然是不安全的。并在在jdk1.8中对HashMap进行了很大的更改，在之前HashMap采用的结构是数组加链表，从1.8开始HashMap的结构变化成数组加链表加红黑树。当链表长度大于8时，链表就会转化成红黑树，提高在Hash碰撞时，提高查询的效率。

有时候我们在使用并发工具库的时候，并不能充分了解他的特性，还是按照老方式使用新工具导致无法发挥其性能。比如，在使用了ConcurrentHashMap时，缺没有充分利用它提供的基于CAS安全的方法，还是使用锁的方式来实现逻辑，这就导致了命名可以减少执行时间的方法而没有去使用。

使用一个场景来模拟普通的一些操作。使用ConcurrentHashMap类，然后创建多个线程，循环一千万次往Map中插入数据，如果key值一样，则加1，最后输出所花费的时间。因为ConcurrentHashMap只能保证读写操作的线程安全，所以在这个例子中，需要在Map进行判断key值是否存在，和value加1操作的外层添加上synchronize锁，来保证线程的安全。为了验证是否确实需要锁来保证安全，在去除锁后，发现value加起来的总值，总是少于预期值。在ConcurrentHashMap提供了原子性方法computeIfAbsent来做复合逻辑操作，判断key是否存在value，如果不存在则把第二个参数的Lanbda表达式运行后的结果放入Map作为value，也就是在表达式中返回一个value，当做初始值设置进map中。最后输出结果，发生大概在1秒左右，大大提升了之前执行的效率。因为在源码的核心部分，computeIfAbsent中使用了Java自带的Unsafe实现的CAS，它在虚拟机层面确保了写入数据的原子性，比加锁的效率高得多。

像ConcurrentHashMap这样的高级并发工具提供了一些高级的API，只有充分了解其特性才能最大化其威力，而不能只是因为说它能够解决什么问题，而盲目的去使用它。

经常我们在使用工具类的时候，没有了解清楚工具的适用场景，在不适合的场景下使用了错误的工具导致性能更差。比如，没有理解CopyOnWriteArrayList的适用场景，把它用在了读写均衡或者大量写操作的场景下，导致性能问题。一般来说，针对通用场景的通用解决方案，在所有场景下性能都还是可以的，属于“万金油”。但针对特殊场景的特殊实现，会有比通用解决方案更高的性能，但一定要在它针对的场景下使用，否则可能会产生性能问题甚至是。

读写分离是一个比较新的技术，不管是在Linux和Redis中都会用到。CopyOnWriteArrayList虽然是一个线程安全的集合，但是因为它实现线程安全的方式的特别，每次修改数据时都会复制一份数据出来，所以就有了明显的适用场景，即读多写少或者说希望无锁读的场景。

编写两段代码分别测试CopyOnWriteArrayList和ArrayList普通加锁方式的读写性能。发现在大量写的情况下（十万次add操作），发现CopyOnWriteArrayList比正常ArrayList加锁操作，慢了将近一百倍。在大量读的场景下（十万次get操作），CopyOnWriteArrayList又比同步的ArrayList快了将近五倍以上。

造成这个原因是，CopyOnWriteArrayList在每次add的时候，都会用Arrays.copyOf创建一个新数组，这个操作就类似于ArrayList每次容量不足时的扩容操作，消耗大量的内存操作，降低系统运行时的性能。所以CopyOnWriteArrayList在频繁add时内存的申请释放消耗比ArrayList大很多。

我们为了保证线程的安全性，经常会使用synchronize加锁的方式去给数据加锁。但在加锁时，应该重点考虑加锁的位置，使它加上之后是否能够保证数据的一致性。

使用一段代码，在一个类里有两个int类型的字段a和b，有一个add方法循环一万次对a和b同时进行++操作，有另一个compare方法，同样循环一万次判断a是否大于b，条件成立时就打印a和b的值。按照正常的逻辑，这两个值应该是始终相当的，但是执行完代码后，发现不但输出了日志，而且输出的a和b的值，有时候b大于a。

出现这个情况的原因是，a > b这个比较操作在字节码层面是加载a、加载b和比较三步，代码虽然是一行但并不是原子性的，可能在比较的途中，数据就发生了变化。一般可能会先考虑在add方法中加锁，但执行后，最后执行结果也是一样的。因为这个add方法始终只有一个线程在操作，显然只为add方法加锁时没用的。所以正确的做法应该是，同时为add和compare都加上方法所，确保add方法执行时，compare无法读取a和b。

我们在加锁时要清楚锁有没有正确锁住被保护的对象，我们要保护的是什么逻辑，多线程执行的情况又是怎么的。比如静态字段属于类，类级别的锁才能保护；而费静态字段属于类实例，实例级别的锁就可以保护。只有清楚的了解了锁住的代码块与逻辑，才能更好的对锁进行应用。

我们在使用锁的情境下时，加锁应该尽可能的考虑粒度和场景，所保护的代码意味着无法进行多线程操作。对于Web类型的天然多线程项目，对方法进行大范围加锁会显著降级并发能力，要考虑尽可能地只为必要的带啊加锁，降低锁的粒度；而对于要求超高性能的业务，还要细化考虑锁的读写场景，以及悲观优先还是乐观优先，尽可能针对明确场景精细化加锁方案，可以在适当的场景下考虑ReentrantReadWriteLock、StampedLock等高级的锁工具。

特务逻辑中有多把锁时要考虑死锁问题，通常的规避方案是，避免无限等待和循环等待。此外，如果有任务逻辑中锁的实现比较复杂的话，我们还要去仔细看看加锁和释放锁是否配对，是否有遗漏释放或重复释放的可能性；并且要考虑所自动超时释放了，而业务逻辑却还在进行的情况下，如果别的线程或进程拿到了相同的锁，可能会导致重复执行。

我们在对方法资源进行加锁时，使用synchronize关键字是最简单的，但是并不能去滥用synchronize，把需要加锁的资源，全部用上synchronize修饰。这种方法是没必要的，通常情况下我们的业务代码是三层架构，数据经过Controller、Service、Repository到数据库，没必要使用synchronize来保护什么数据。而且这可能会极大地降低性能，在使用Spring时，通常下这三层的类都是单例的，加上synchronize会导致整个程序几乎就只能支持单线程，造成极大的性能问题。

在程序中，我们会使用到各种池化的技术来缓存创建昂贵的对象，比如线程池和数据库连接池等。由于线程的创建比较昂贵，随意、没有控制地创建大量的线程会造成性能问题，因此短平快的任务一般考虑使用线程池来处理，而不是直接创建线程。

虽然在使用线程池时，Java中的Executor类定义了一些快捷的工具方法，来帮助我们快速创建线程池。但观看《阿里巴巴Java开发手册》中提到了，禁止使用这些内置的方法来创建线程池，而应该手动new ThreadPoolExecutor的方法来创建线程池。其中最典型的就是Executor中的newFixedThreadPool和newCacheThreadPool，可能因为资源耗尽导致OOM问题。

为了了解出现这个情况的原因，查看了这两个方法中的实现。观看源码时，发现newFixedThreadPoolExecutor中的等待工作对待直接使用了new一个LinkedBlockingQueue，而LinkedBlockingQueue的默认构造方法是一个Integer.MAX\_VALUE长度的队列，可以认为是无界的。也就是说，线程只要进来，不管多少，并不会触发设置的拒绝策略，全部在这个队列中进行等待。如果任务较多并且执行较慢的话，队列可能会快速挤压，撑爆内存导致OOM。

继续去测试使用newCachedThreadPool创建线程池的情况。发现它创建的线程池的最大线程数也是Integer.MAX\_VALUE,而其等待队列是一个没有存储空间的阻塞队列。这也就是说，只要有请求到来，就必须找到一个工作线程来处理，如果没有就再创建一个新的线程去执行。如果线程池的任务因为而卡主时，这就会导致系统中存在大量的阻塞线程，致使系统抛出异常并崩溃。

今天继续学习Java线程池的各种应用。为了更加直观的查看线程池的基本特性，编写了一段程序进行测试。首先定义了一个线程池，这个线程池具有2个核心线程、5个最大线程、使用容量为10的阻塞队列作为工作队列，使用默认的AbortPolicy拒绝策略，也就是任务添加失败的会抛出一个异常。然后每隔1秒向线程池提交任务，循环20次，每隔任务让他在执行的时候，休眠10秒钟。并且使用一段代码来观察线程池运行时候的各种状态。

运行代码后，观察线程池中的状态，总结出了一下线程池默认的工作行为：不会初始化核心线程数，只有任务来了才创建工作线程；当核心线程满了之后不会立即扩容线程池，而是把任务堆积到工作队列中；当工作队列满了后扩容线程池，一直到线程个数达到maxinumPoolSIze为止；如果队列已满且达到了最大线程后还是有任务进来，按照拒绝策略处理；当线程数大于核心线程数时，线程等待keepAliveTime后还是没有任务需要处理的话，收缩线程到核心线程数。

在了解了线程池的这个策略后，对线程池更项参数的设置，有了更好的理解，以便为线程池设置合适的初始化参数。同时也可以通过一些手段来改变这些默认的工作行为。比如声明线程池后立即调用prestartAllCoreThreads方法，直接启动所有的核心线程数。传入true给allowCoreThreadTimeOut方法，来让线程池在空闲的时候同样回收核心线程。

学习了线程池会后，接着学习对连接池的使用。一般连接池的结构是对外提供获得连接、归还连接的接口给客户端使用，并暴露最小空闲连接数、最大连接数等可配置参数，在内部则实现连接建立、连接心跳保持、连接管理、空闲连接回收、连接可用性检测等功能。

我们在使用第三方进行网络通信时，一定要注意是否是基于连接池的。因为TCP是面向连接的基于字节流的协议，面向连接，意味着连接需要先创建再使用，创建连接的三次握手有一定的开销；基于字节流，意味着字节是发送数据的最小单位，TCP协议本身无法区分哪几个字节是完整的消息体，也无法感知是否有多个客户端在使用同一个TCP连接，TCP只是一个读写数据的管道。如果第三方没有使用连接池的话，而直接是TCP连接，那么就需要考虑每次简历TCP连接的开销，并且因为TCP是基于字节流，在多线程的情况下对同一连接进行复用，可能会产生线程安全问题。

在学习了SDK连接池的实现的几种形式后，大概整理出了对连接池的最佳实践：如果是连接池和连接分离的方式，那么连接池一般是线程安全的，可以复用。每次使用需要从连接池获取连接、使用后归还，归还的工作由使用者负责。如果是内置连接池，SDK会负责连接的获取和归还，使用的时候直接复用客户端；如果SDK没有实现连接池（大部分中间件、数据库的客户端SDK都会支持连接池），那通常不是线程安全的，而且短连接的方式性能不会很高，使用的时候需要考虑是否自己封装一个连接池。

为了验证昨天学习的连接实现的方式分类，今天学习使用Jedis类对redis进行操作，分析Jedis是处于哪一种连接分类，与它所能够保证的情况。

现在redis中初始化两组数据，a=1，b=2。然后启动了两个线程，使用同一个Jedis实例进行操作，每一个线程循环一万次，分别读取key为a和b的value，判断是否分别为1和2。执行结束后，观看控制台的结果发现，key值和value值混乱，并且还会抛出各种异常，Socket被关闭等。

跟着书里的介绍一起去观看Jedis的源码实现，Jedis最终的实现方法最终会调用Connection类中的方法，使用Protocol类发送命令。在Protocol类中发现它发送命令是直接操作RedisOutput写入字节。所以这也就导致了如果多个线程在执行操作，那么即无法确保整条命令以一个原子操作写入Socket，也无法确保写入后，读取前没有其它数据写到远端。在这个例子中，多个线程使用同一个Jedis实例，写操作互相干扰，多条命令相互穿插的话，必然不是合法的redis命令，那么redis会关闭客户端连接，导致连接断开。刚才线程1和2先后写入了get a和get b操作的请求，redis也是正确返回了值，但是线程2先读取了线程1就会出现数据错乱的问题。

修复的方式是，使用Jedis提供的另一个线程安全的类JedisPool来获得Jedis的实例。JedisPool可以声明在static中给多个线程之间共享，扮演连接池的角色。由此可以看出，Jedis可以单独使用，也可以配合连接池使用，所以他是属于连接池和连接分离的实现方式。

我们在使用池化技术的时候，一定要注意对池的复用，否则他的使用代价会比每次创建单一对象更大，对连接池来说也是一样的。总结了一下几点原因：

创建连接池的使用很可能一次性创建了多个连接，大多数连接池考虑到性能，会在初始化的时候维护一定数量的最小连接。但是每次使用使用连接池都按需创建连接池，那么很可能只用到一个连接，但是创建了N个连接。连接池一般会有一些管理模块，限制连接有独立线程管理，启动了空闲检测的连接池相当于还会启动一个线程。此外，有些连接池还需要独立线程负责连接保护等功能。因此，启动一个连接池相当于启动了N个线程。

我们对连接池的配置并不是一层不变的，各个配置项都应该跟着自己项目中的实际运行参数和场景进行设置。其中最重要的参数是最大连接数，它决定了连接池能使用的连接数量上限，达到上限后，新来的请求需要等待其他请求释放连接。但最大连接数并不是设置的越大越好，如果设置得太大，客户端需要耗费过多的资源维护连接，更重要的是由于服务端对应的是多个客户端，每一个客户端都保持大量的连接，这便会给服务端带来巨大的压力。当然最大连接数也并不是越小越好，很可能会因为获取连接的等待时间过长，导致吞吐量底下，甚至超时无法获取连接。只有合理的设置连接池的参数，才能够保证性能的最大优化。

环境的配置的参数是至关重要的，这些参数的配置一般来说会很大程度影响系统执行时的效率。我们在使用Http协议进行一次网络请求时，网络请求必然会有超时的可能性。所以对此我们就要考虑到这几点，框架设置的默认超时时间是否合理；考虑到网络的不稳定，超时后的请求重试的次数；框架是否会像浏览器那样限制并发连接数，以免在服务并发很大的情况下，HTTP调用的并发数限制成为瓶颈。

一般对请求超时，会两个重要的配置，连接超时和读取超时。这两个参数有时候很难引起我们的注意，但是正确理解和配置这两个参数还是比较有意义的。有可能我们在开发时，也会遇到两个误区：

连接超时时间配置得特别长，比如60秒。一般来说，TCP建立连接需要的时间是非常短的，通常在毫秒级最多到秒级，不可能需要十几秒甚至几十秒。如果几秒连接不上，那么很可能永远也连接不上，就不需要继续等待，以此来耗费时间。

排查连接超时问题时，却没理清连的是哪里。通常情况下，服务会有多个节点，如果别的客户端通过客户端负载均衡技术来连接服务端，那么客户端和服务端会直接建立链接，此时出现连接超时大概率是服务端的问题，此时出现连接超时大概率是服务端的问题；而如果服务端通过类似Nginx的方向代理来负载均衡，客户端连接的气势是Nginx，而不是服务端，此时出现连接超时应该排查Nginx。

Spring的声明式事务功能为我们提供了极其方便的事务配置方式，配合Spring Boot的自动配置，大多数Spring Boot项目只需要在方法上标记@Transaction注解，即可意见开启方法的事务性配置。

但是我们在使用时也需要小心Spring的事务可能没有生效，@Transaction要想让事务生效，他有几种必须遵循的准则：第一点，配置特殊的配置（比如使用AspectJ实现aop），否则只有定义在public方法上的事务直接才能生效，Spring默认通过动态代理的方式实现AOP，对目标方法进行增强，private方法无法代理到，Spring自然也无法动态增强事务处理逻辑。第二点，必须通过代理过的类从外部调用目标方法才能生效。Spring通过AOP技术对方法进行增强，要调用增强过的方法必然是调用代理后的对象。

有时候我们事务即便生效也不一定能回滚。通过AOP实现事务处理可以理解为，使用try catch来包裹标记了@Transaction注解的方法，当方法出现了异常并且满足一定条件的时候，在catch里面我们可以设置事务回滚，没有异常则直接提交事务。所以，只有异常传播出了标记了@transaction方法，事务才能回滚。默认情况下，出现RuntimeException（非收检异常）或Error的时候，Spring才会回滚事务。

我们也可以自己捕获异常后，可以手动设置让当前事务处于回滚状态。调用TransactionAscpectSupport.currentTransactionStatus().setRollbackOnly()方法来让事务进行回滚。可以在注解中声明，期望遇到所有的Exceotion都回滚事务，以此来突破默认不回滚受检异常的限制。

在考虑到数据库查询的效率时，一般都会增加索引来进行快速查询，但索引也并不是万能的。我们可能也会陷入到索引的几个误区中：

考虑到索引的维护代价、空间占用和查询时会标的代价，不能认为索引越多越好。索引一定是按需创建的，并且要尽可能确保足够轻量。一旦创建了多个字段的联合索引，我们要考虑尽可能利用索引本身完成数据查询，减少回表的成本；不能认为创建了索引就一定有效，对于后缀的匹配查询、查询中不包含联合索引的第一列、查询条件设计函数计算等情况无法使用索引。此外，即使SQL本身符合索引的使用条件，Mysql也会用过评估各种查询方式的代价，来决定是否走索引，以及走哪个索引。

有时候会导致索引时候的情况，索引只能匹配前列前缀。有时候在使用LIKE语句，开头的时候写了%进行匹配，数据库使用了全表扫描。而将%放在最后匹配，数据库就会使用索引进行查找；比如有时候使用了数据库自带的函数进行查找，它也是不能够走索引的。

因此，我们在尝试通过索引进行SQL性能优化的时候，一定要通过执行计划或实际效果来确定索引是否能够有效改善性能的问题，否则增加了索引不但没解决性能问题，还增加了数据库的负担。在测试时，发现EXPLAIN给出的执行计划有疑问的话，还可以利用optimizer\_trace查看详细的执行计划。

在Java的高并发类中，有很多工具使用到了CAS的技术。为了对其更好的理解，在网上查阅了各项资料，对CAS的实现进行学习，以便更好的对工具进行使用。发现CAS的全称为Compare-And-Swap（比较并交换），它是一条cpu并发原语。他的功能时判断内存某个位置的值是否为预期值，如果是预期值则更改为最新的值，这个过程是原子性的，不可中断。

CAS的底层原理，Unsafe是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地（native）方法来访问，Unsafe相当于一个后门，基于该类可以直接操作特定内存的数据。其内部方法操作可以像C的指针一样直接操作内存。Unsafe类中的所有方法都是native修饰的，也就是Unsafe类中的方法都直接调用操作系统底层资源执行相应任务

CAS的乐观锁思想虽然执行效率在有的情况下比悲观锁快，但同样也有着它的缺点。CAS的缺点。循环时间长开销大，因为在进行修改操作时，是使用自旋锁进行判断主内存是不是自己所期望的预期值，不是的话则一直做循环操作，直至主内存达到自己预期值的要求。因为如果一直判断不成功就一直循环，占用CPU的资源；只能保证一个共享变量的原子操作；可能会造成出现ABA问题。

ABA产生的问题是，CAS算法的实现有一个重要的前提是需要取出内存中某时刻的数据并在当下时刻比较并替换，那么在这个时间差内有可能会导致数据的变化。CAS只判断存取前的数据是否相等，没有去判断中间的时候值是否已经被更改。所以在修改的时候添加判断条件，如版本号、时间戳等方式进行解决。

在编写项目的实体时，有许多的构造方法、getter和setter方法需要编写。这个过程一般来说是比较机械、繁琐的。学习了Lombok之后，对这个问题很好的解决了。为了学习，先去他的官网看了下他的介绍，在引入了Lombok之后，对象的构造函数、equal()方法，属性的get/set方法等等，都可以让Lombok进行解决。

以前我们在写一个POJO对象时，我们可以借助与Lombok，一切将会变得十分简单，可以在属性的定义上添加@Getter和@Setter方法，这时候代码就会隐式的设置每一个属性的获取和设置的方法。我们以前在创建一个对象的时候，往往是需要new一个对象，然后一行一行的设置他的属性，这种代码虽然可阅读性比较好，但是写的比较繁琐，占用较多的代码行数，不够简洁。这时候就可以在类上添加@Builder注解，在创建的时候就可以使用链式表达的方式一行代码完成，相当于设计模式中的“建造者模式”。其中它还包含了许多注解方法，例如：

@AllArgsConstructor：添加全参数的构造方法

@NoArgsConstructor：添加无参的构造方法

@RequiredArgsConstructor：添加部分参数的构造方法

@NonNull：添加到参数列表的参数前面，可以对参数进行自动提前的非空判断

@Cleanup：放在属性定义的前面，可以自动关闭各项连接，降低资源泄露的风险

@SneakyThrows：添加在方法前面，就可以不用处理异常

Lombok也有很强的弊端，在使用了Lombok后，其它的开发人员也必须使用安装他的插件，否则项目就会抛出错误。而且Lombok都是隐式的设置类那些代码，所以也就增加了排查代码错误的难度。