有了解过 springboot 吗

四大核心

1. 自动配置
2. 起步依赖
3. Actuator
4. 命令行界面

SpringBoot 重要策略

1. 开箱即用
2. 约定优于配置

SpringBoot 特性

1. 能够快速创建基于 Spring 的应用程序，避免了大量的配置文件
2. 能够直接使用 java main 方法启动内嵌的 Tomcat 服务器运行 SpringBoot 程序，不需要部署 war 包文件
3. 提供约定的 starter POM 来简化 Maven 配置，让 Maven 的配置变得简单
4. 自动化配置，根据项目的 Maven 依赖配置，SpringBoot 自动配置 Spring、SpringMVC 等
5. 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标，健康检测、外部配置等
6. 基本可以完全不使用 XML 配置文件，采用注解配置
7. SpringBoot 不是对 Spring 功能上的增强，而是提供了一种快速使用 Spring 的方式，简单理解，就是框架的框架

Springboot 启动注解

@SpringBootApplication

Springboot 自动装配原理

1. @SpringBootApplication是个复合注解，分别由@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan三个注解构成，包含了这三个注解的功能
2. @ComponentScan用来扫描指定包下的注解，默认是启动类的同级或者子级包，可以手动配置其默认的包扫描路径
3. @SpringBootConfiguration注解上包含了@Configuration注解，表明当前类是一个配置类
4. @EnableAutoConfiguration代表SpringBoot 自动配置功能开启，所有的自动装配功能都在这个注解上
5. @EnableAutoConfiguration注解是一个合成注解，包含了@AutoConfigurationPackage和@Import(AutoConfigurationImportSelector.class)
6. @AutoConfigurationPackage代表自动配置包，内部调用了@Import(AutoConfigurationPackages.Registrar.class) 将Registrar这个类注册到Spring容器中，在这里完成了所有类的自动装配操作
7. @Import(AutoConfigurationImportSelector.class)用来获取所有的配置类并注入到Spring容器中
8. 其中，SpringFactoriesLoader.loadFactoryNames 方法的作用就是从META-INF/spring.factories文件中读取指定类对应的类名称列表，所有的类就是从这里加载的。即所有自动装配的类都是由autoconfigure包下的META-INF/spring.factories配置文件中加载的
9. 所有配置类都是通过@Configuration以及各种@Conditional条件装配进行注入的。如果满足条件就注入，不满足就不会注入，甚至还有默认值选项（约定大于配置）以及@EnableConfigurationProperties导入自定义配置文件中的配置

HashMap 怎么遍历

```
private final Map<String, Object> map = new HashMap<>();

@Before
public void init() {
    map.put("id", "易烊千玺");
```

```
    map.put("name", "迪丽热巴");
    map.put("age", "古力娜扎");
    map.put("info", "马尔扎哈");
}
```

1. 获取所有的键，根据键获取对应的值

```
@Test
public void test1() {
    Map<String, Object> map = new HashMap<>();

    map.put("id", "易烊千玺");
    map.put("name", "迪丽热巴");
    map.put("age", "古力娜扎");
    map.put("info", "马尔扎哈");

    // 获取所有的键
    Set<String> set = map.keySet();

    // 遍历所有的键
    for (String key : set) {
        // 根据键获取对应的值
        System.out.println(key + ":" + map.get(key));
    }
}
```

2. 获取 entrySet 并迭代拿到每一个 entry 即可拿到所有的键和值

```
@Test
public void test2() {
    // 获取entrySet
    Set<Map.Entry<String, Object>> entrySet = map.entrySet();

    // 遍历entry获取所有entry
    for (Map.Entry<String, Object> entry : entrySet) {
        // 获取entry中的键和值
        System.out.println(entry.getKey() + ":" + entry.getValue());
    }
}
```

3. 获取迭代器，再通过迭代器获取 entry【等同于第二种方式，不建议使用】

```
@Test
public void test3() {
    // 获取entrySet
    Set<Map.Entry<String, Object>> entrySet = map.entrySet();

    // 获取迭代器
    Iterator<Map.Entry<String, Object>> iterator = entrySet.iterator();

    // 判断迭代器是否还有下一个元素
    while (iterator.hasNext()) {
        // 获取entry
        Map.Entry<String, Object> entry = iterator.next();

        System.out.println(entry.getKey() + ":" + entry.getValue());
    }
}
```

4. 分别获取所有的键以及所有的值进行遍历

```java
@Test
public void test4() {
    // 获取所有的键
    Set<String> keys = map.keySet();

    for (String key : keys) {
        System.out.println(key + ":");
    }

    // 获取所有的值
    Collection<Object> values = map.values();

    for (Object value : values) {
        System.out.println(value);
    }
}
```

SpringMvc 的工作流程

1. 用户发送请求至前端控制器 DispatcherServlet。2. DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。
2. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。
3. DispatcherServlet 通过 HandlerAdapter 处理器适配器调用处理器
4. 执行处理器(Controller，也叫后端控制器)。
5. Controller 执行完成返回 ModelAndView
6. HandlerAdapter 将 Controller 执行结果 ModelAndView 返回给DispatcherServlet
7. DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器
8. ViewReslover 解析后返回具体View
9. DispatcherServlet 对 View 进行渲染视图（即将模型数据填充至视图中）。
10. DispatcherServlet 响应用户

Controller 怎么返回 JSON 数据

1. 导入 Jackson 或者 FastJson 依赖，控制层方法上使用 @ResponseBody 注解，返回值为 Map 或者对象
2. 导入 Jackson 或者 FastJson 依赖，控制层方法把对象解析成 JSON 字符串，设置 response 响应类型为 application/json 并使用 getWriter().append() 返回 JSON 字符串

创建线程的方式

1. 自定义线程类，继承自 Thread，并重写 run 方法
2. 自定义线程类，传入一个实现了 Runnable 接口的参数【重点】
3. 自定义线程类，传入一个实现了 Callable 接口的参数

JSP 执行完生成什么文件

.java 文件，因为 JSP 底层实际上就是 Servlet

SpringBoot 怎么打包的，打包完怎么执行

声明打包方式为 jar（默认），使用 mvn package 命令完成打包，打包完使用 java -jar 命令后面跟上 jar 包的名称即可运行。运行得到的效果和使用 idea 是一样的。

MySQL 分库分表了解过没有

MyCat 和 Sharding Sphere（就是Sharding Jdbc）

SpringCloud 了解过没有

微服务架构

微服务架构样式是一种将单个应用程序开发为一组小服务的方法，每个小服务都在自己的进程中运行并与轻量级机制（通常是HTTP资源API）进行通信。这些服务围绕业务功能构建，并且可以由全自动部署机制独立部署。这些服务的集中管理几乎没有，它可以用不同的编程语言编写并使用不同的数据存储技术。

1. 微服务架构只是一个样式，一个风格。
2. 将一个完成的项目，拆分成多个模块去分别开发。
3. 每一个模块都是单独的运行在自己的容器中。
4. 每一个模块都是需要相互通讯的。 Http，RPC，MQ。
5. 每一个模块之间是没有依赖关系的，单独的部署。
6. 可以使用多种语言去开发不同的模块。
7. 使用MySQL数据库，Redis，ES去存储数据，也可以使用多个MySQL数据库。

总结：将复杂臃肿的单体应用进行细粒度的划分，每个拆分出来的服务各自打包部署。

Spring Cloud 概述

1. SpringCloud 是微服务架构落地的一套技术栈。
2. SpringCloud 中的大多数技术都是基于 Netflix 公司的技术进行二次研发。
3. SpringCloud 中文社区网站：http://springcloud.cn/
4. SpringCloud 中文网：http://springcloud.cc/

Spring Cloud 核心组件

- **Eureka** - 服务的注册与发现
- **Robbin** - 服务之间的负载均衡
- **Feign** - 服务之间的通讯
- **Hystrix** - 服务的线程隔离以及断路器
- **Zuul** - 服务网关
- Stream - 实现MQ的使用
- Config - 动态配置
- Sleuth - 服务追踪

消息中间件了解过没有

> 消息中间件指的就是消息队列（MQ），就是在生产者和消费者中间添加一个中间件（容器），用来控制消息的转发和存储。生产者先将消息投递一个叫做「队列」的容器中，然后再从这个容器中取出消息，最后再转发给消费者，仅此而已。
>
> 比如我们平时的超市购物中，当我们在结算的时候，并不会一窝蜂一样涌入收银台，而是排队结算。这也是队列机制，就是排队。一个接着一个的处理，不能插队。

为什么要使用消息队列

解耦

比如说我有一个系统，A模块需要调用BCD的模块去给它们发送数据，如果加入了新的模块，或者有的模块不需要接收数据了，修改代码会变得很繁琐，并且还要考虑某个模块挂掉了或者消息重发的问题！

如果我用了 MQ，那么我的A模块只需要把消息发送到 MQ 里面去，哪个模块需要调用就直接来监听消费即可，这样的解耦就能避免了A模块修改代码，也不需要考虑是否调用成功，请求超时等问题了。（发布订阅模式）

异步

比如说我有一个场景，A模块执行需要一个请求需要调用BCD三个接口后返回给用户，这个过程是同步高延时的，必须一个一个接口调用，时间是累加的，而且如果任何一个模块出现了问题都会导致后面的模块无法继续进行。

如果我使用了 MQ，我只需要把消息发送给三个消息队列中就可以直接返回给用户，用户的感受就是非常快，在等待用户下次请求之前BCD三个模块都从各自监听的队列中消费数据即可，对用户而言这个时间是不存在的，体验会很好

削峰

比如说我有一个高并发的场景，每秒有5000个请求过来，都直接访问数据库很可能直接把数据库打死。

如果我使用了 MQ，那么我就可以通过 MQ 每秒发送2000个请求给数据库，剩余的请求就放在 MQ 中缓存起来，后面再慢慢发，当高峰期过了 MQ 中的请求就能处理完成

引入 MQ 后可能存在的问题

1、系统的可用性降低，如果 MQ 挂掉了，那么A 系统就没有办法往MQ中发送消息，BCD也没有办法从MQ中获取消息，导致整个系统崩溃

2、系统的复杂性提高，需要考虑 MQ 的很多问题，比如重复发送，重复消费，消息丢失，消息顺序乱掉等问题

3、原子性问题，可能我A发送的消息需要BCD三个都成功后才能返回给用户，但是我中间有一个失败了，其实整个请求都应该失败，但是用户收到的确实请求成功

消息队列的优缺点

上面两个总结一下

ActiveMQ、RabbitMQ、RocketMQ、Kafka的区别

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，吞吐量比 RocketMQ 和 Kafka 要低一个数量级	万级，吞吐量比 RocketMQ 和 Kafka 要低一个数量级	10万级，RocketMQ 也是可以支撑高吞吐量的一种 MQ	10万级，这是 kafka 最大的优势，就是吞吐量高 一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百，几千个的级别，吞吐量会有较小幅度的下降 这是 RocketMQ 的一大优势，在同等机器下，以支持大量的topic	topic 从几十个到几百个的时候，吞吐量会大幅度下降，支撑大规模topic，需要增加更多的机器资源
时效性	ms级	微秒级，这是 RabbitMQ的一大特点，延迟是最低的	ms级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用性	高，基于主从架构实现高可用性	非常高，分布式架构	非常高，kafka是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到0丢失	经过参数优化配置，可以做到0丢失

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
核心特点	**MQ 领域的功能及其完备**	**基于 erlang 开发，所以并发能力很强，性能及其好，延时很低**	**MQ 功能较为完善，还是分布式的，扩展性好**	**功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准**
优劣势总结	非常成熟，功能强大，在业内大量的公司以及项目中都有应用 偶尔会有较低概率丢失消息，而且现在社区以及国内应用都越来越少，官方社区现在对ActiveMQ维护越来越少，而且确实主要是基于解耦和异步来用的，较少在大规模吞吐的场景中使用	erlang 语言开发，性能及其好，延时很低；而且开源提供的管理界面非常棒，用起来很好用，在国内一些互联网公司近几年用RabbitMQ也比较多一些 但是问题也是显而易见的，RabbitMQ确实吞吐量会低一些，这是因为他做的实现机制比较重。而且 erlang 开发，国内有几个公司有实力做 erlang源	接口简单易用，而且毕竟在阿里大规模应用过，有阿里品牌保障 日处理消息上百亿之多，可以做到大规模吞吐，性能也非常好，分布式扩展也很方便，社区维护还可以，可靠性和可用性都是ok的，还介意支撑大规模的topic数量，支持复杂MQ业务场景	kafka的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展 同时kafka最好是支撑较少的topic数量即可，保证其超高吞吐量 而且kafka唯一的一点劣势是有可能消息重复消费

如何选择MQ

中小型公司，实力不太足就用RabbitMQ就行，社区比较活跃，管理页面用起来很方便，能够支撑高并发

大型公司，实力比较雄厚，能够对底层源码进行定制，可以使用RocketMQ，吞吐量比 RabbitMQ 高一个量级，分布式架构扩展方便

大数据领域直接 kafka

通过 explain 查看是否走索引，看哪里判断是否走索引

explain 后直接加上SQL语句即可，生成的结果中 POSSIBLE_KEYS 字段代表能够使用的索引，key 字段对应实际使用的索引，KEY_LEN 字段代表索引的长度

```
1 explain select * from user where id = 1;
```

信息 | 结果 1 | 剖析 | 状态

id	select_type	table	partitio	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	const	PRIMARY	PRIMARY	4	const	1	100	(Null)

> 补充：执行计划中的所有内容概述

字段	描述
id	代表SQL语句的执行顺序。ID值相同时，说明SQL执行顺序是按照显示的从上至下执行的；ID值不同时，ID值越大代表优先级越高，则越先被执行
select_type	查询的类型
table	表名，或者表的别名
partitions	分区

字段	描述
type	查询的性能。const 效率最高，all 效率最低
possible_keys	指出MySQL能使用哪些索引来优化查询（不一定会被使用）
key	查询实际所使用的索引
key_len	显示MySQL索引所使用的字节数
ref	表示当前表在利用Key列记录中的索引进行查询时所用到的列或常量
rows	查询所需的行数（不一定准确）
filtered	表示返回结果的行数占需读取行数的百分比，值越大越好（值越大，表明实际读取的行数与所需要返回的行数越接近）。同样不一定准确
extra	额外的附加信息

事务的隔离级别

1. 读未提交（Read uncommitted）：什么都不能保证
2. 读已提交（Read Committed）：可避免脏读情况发生
3. 可重复读（Repeatable Read）：可避免脏读、不可重复读情况的发生
4. 序列化（Serializable）：可避免脏读、不可重复读、虚读情况的发生

【注意】事务的隔离级别越高，安全性越高，效率越低！！！

接口和抽象类的区别

1. 抽象类是类，只能被子类单继承；接口可以被其他接口继承，并且是多继承，还能被多实现。
2. 抽象类中可以有非抽象方法（成员方法和构造方法）；接口中只有抽象方法（JDK1.8 新增 default 方法和 static 方法）

Java 中截取字符串的方法

```
// 返回一个字符串，该字符串是此字符串的子字符串。子字符串从指定索引处的字符开始，并延伸到该字符串的末尾。
String substring(int beginIndex, int endIndex)
```

MySQL 中去重的方式

distinct

MyBatis 中的二级缓存

1. MyBatis中有两级缓存，第一级是 SqlSession 级别的缓存，无需配置自动开启
2. 第二级是SqlSessionFactory 级别的缓存，其中有一个核心概念为缓存命中率
3. 开启二级缓存需要在 在对应的 mapper.xml 配置文件中声明 cache 标签，并且将映射结果对应的实体类实现序列化接口

MyBatis 中的两种结果集

1. 进行 select 映射的时候，返回类型可以用 resultType，也可以用 resultMap，resultType 直接表示返回类型，使用的格式为完整的全限定名；而 resultMap 则是对外部 resultMap 的引用，resultType 和 resultMap 不能同时存在
2. 当提供的返回类型属性是 resultType 时，MyBatis 会将 Map 里面的键值对取出赋给 resultType 所指定的对象对应的属性。所以其实 MyBatis 的每一个查询映射的返回类型都是 ResultMap，只是当提供的返回类型属性是 resultType 的时候，MyBatis 对自动的给把对应的值赋给 resultType 所指定对象的属性。

MySQL 中常用的搜索引擎

1. Myisam 存储引擎：每个Myisam在磁盘上存储成三个文件。文件名都和表名相同，扩展名分别为.frm（存 储表定义）、.MYD（存储数据）、.MYI（存储索引）。数据文件和索引文件可以放置在不同目录，平均分布io，获得更快的速度。对存储大小没有限 制，MySQL数据库的最大有效表尺寸通常是由操作系统对文件大小的限制决定的。
2. InnoDB存储引擎：具有提交、回滚、奔溃恢复能力的**事务**安全。与Myisam相比，InnoDB的写效率差一些并且会占用更多的磁盘空间以保留数据和索引。除此之外还支持**行锁**和**外键**。MySQL 5.5后的存储引擎默认为InnoDB。

Redis 持久化方式

1. RDB：在指定的时间间隔内将内存中的数据集快照写入磁盘，数据恢复时将快照文件直接再读到内存。
   i. RDB 文件是二进制文件，不能被查看和修改，但是写入和读取的速度很快，并且非常容易备份
   ii. 灵活性更高，可以手动指定持久化的触发条件
   iii. 具有一定安全性问题（不满足持久化的触发条件时宕机会丢失数据）
2. AOF：Redis 每次接收到一条改变数据的命令时，它将把该命令写到一个 AOF 文件中（只记录写操作，读操作不记录），当 Redis 重启时，它通过执行 AOF 文件中所有的命令来恢复数据。
   i. 对于相同的数据集来说，AOF 文件要比 RDB 文件大，并且是文本文件，虽然可以被查看和修改，但是读取和存储的速度要慢很多。
   ii. 当 AOF 文件变得过大时，会自动地在后台对 AOF 进行重写
   iii. 根据所使用的持久化策略来说，AOF 的速度要慢于 RDB，但是安全性要高。

索引会不会失效，什么情况下会失效