

5.1 Spring面试题总结

作者：Guide哥。

介绍: Github 70k Star 项目 [JavaGuide](#) (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

这篇文章主要是想通过一些问题，加深大家对于 Spring 的理解，所以不会涉及太多的代码！这篇文章整理了挺长时间，下面的很多问题我自己在 Spring 的过程中也并没有注意，自己也是临时查阅了很多资料和书籍补上的。网上也有一些很多关于 Spring 常见问题/面试题整理的文章，我感觉大部分都是互相 copy，而且很多问题也不是很好，有些回答也存在问题。所以，自己花了一周的业余时间整理了一下，希望对大家有帮助。

5.1.1. 什么是 Spring 框架？

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。Spring 官网：<https://spring.io/>。

我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：Core Container 中的 Core 组件是 Spring 所有组件的核心，Beans 组件和 Context 组件是实现 IOC 和依赖注入的基础，AOP 组件用来实现面向切面编程。

Spring 官网列出的 Spring 的 6 个特征：

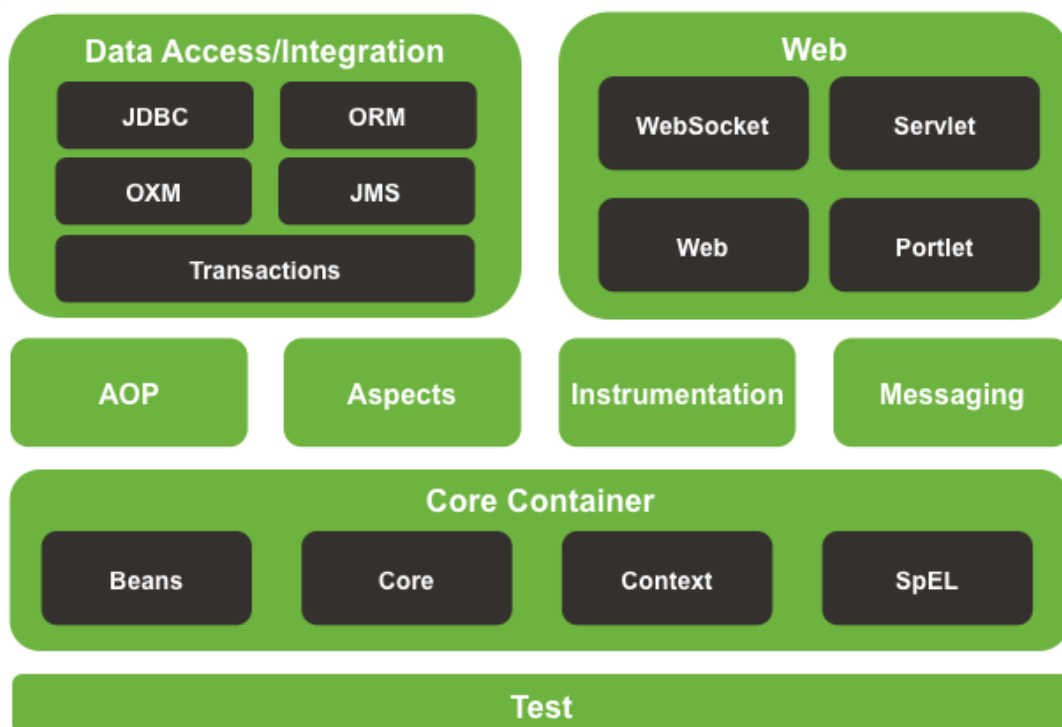
- **核心技术**：依赖注入(DI)，AOP，事件(events)，资源，i18n，验证，数据绑定，类型转换，SpEL。
- **测试**：模拟对象，TestContext 框架，Spring MVC 测试，WebTestClient。
- **数据访问**：事务，DAO 支持，JDBC，ORM，编组 XML。
- **Web 支持**：Spring MVC 和 Spring WebFlux Web 框架。
- **集成**：远程处理，JMS，JCA，JMX，电子邮件，任务，调度，缓存。
- **语言**：Kotlin，Groovy，动态语言。

5.1.2 列举一些重要的 Spring 模块？

下图对应的是 Spring 4.x 版本。目前最新的 5.x 版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。



Spring Framework Runtime

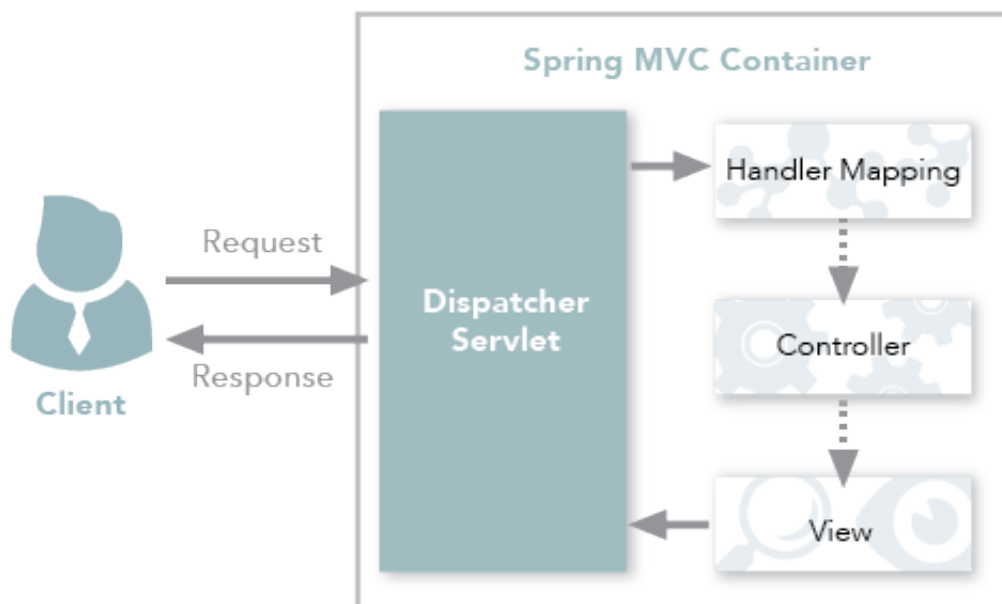


- **Spring Core**: 基础,可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IoC 依赖注入功能。
- **Spring Aspects**: 该模块为与 AspectJ 的集成提供支持。
- **Spring AOP**: 提供了面向切面的编程实现。
- **Spring JDBC**: Java 数据库连接。
- **Spring JMS**: Java 消息服务。
- **Spring ORM**: 用于支持 Hibernate 等 ORM 工具。
- **Spring Web**: 为创建 Web 应用程序提供支持。
- **Spring Test**: 提供了对 JUnit 和 TestNG 测试的支持。

5.1.3 @RestController vs @Controller

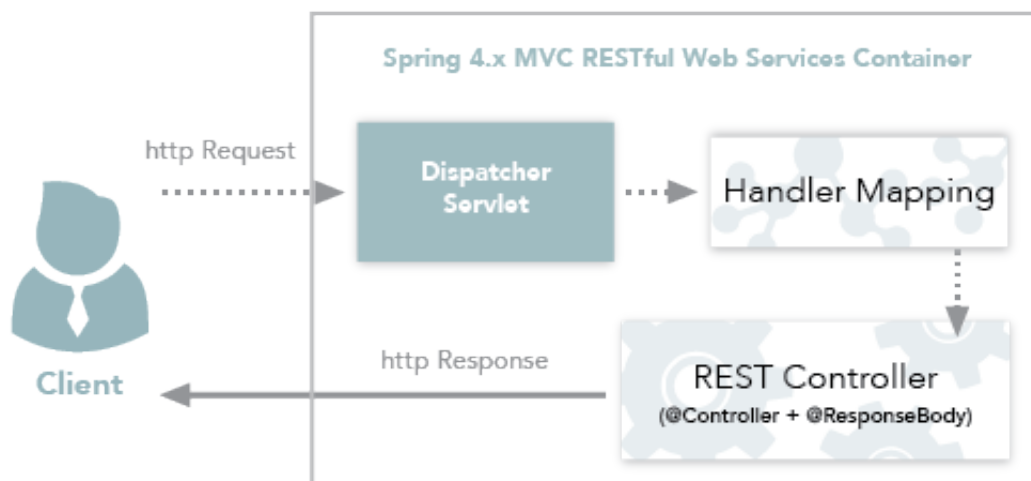
Controller 返回一个页面

单独使用 `@Controller` 不加 `@ResponseBody` 的话一般使用在要返回一个视图的情况,这种情况属于比较传统的 Spring MVC 的应用,对应于前后端不分离的情况。



@RestController 返回JSON 或 XML 形式数据

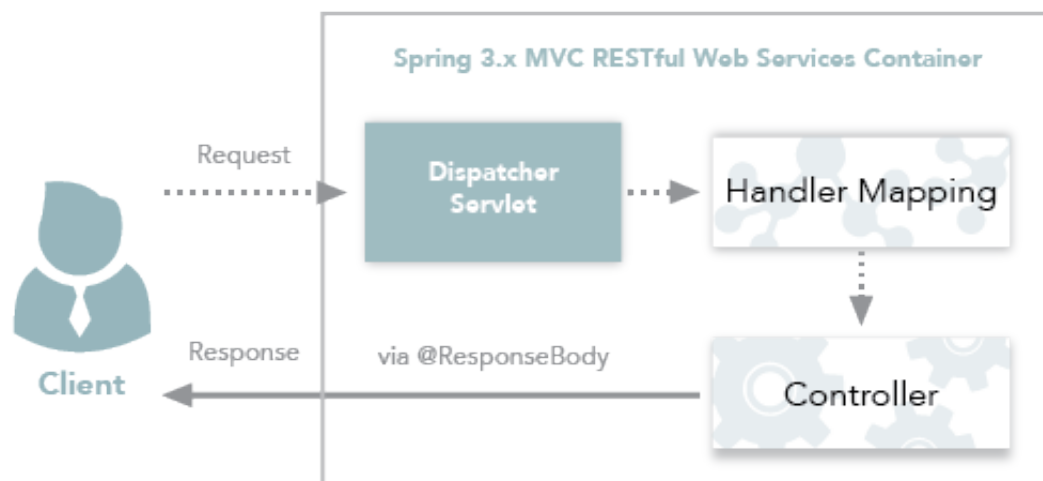
但 **@RestController** 只返回对象，对象数据直接以 JSON 或 XML 形式写入 HTTP 响应 (Response)中，这种情况属于 RESTful Web服务，这也是目前日常开发所接触的最常用的情况（前后端分离）。



@Controller +@ResponseBody 返回JSON 或 XML 形式数据

如果你需要在Spring4之前开发 RESTful Web服务的话，你需要使用 **@Controller** 并结合 **@ResponseBody** 注解，也就是说 **@Controller + @ResponseBody = @RestController**（Spring 4 之后新加的注解）。

@ResponseBody 注解的作用是将 Controller 的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到HTTP 响应(Response)对象的 body 中，通常用来返回 JSON 或者 XML 数据，返回 JSON 数据的情况比较多。



Reference:

- <https://dzone.com/articles/spring-framework-restcontroller-vs-controller> (图片来源)
- <https://javarevisited.blogspot.com/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html?m=1>

5.1.4 Spring IOC & AOP

谈谈自己对于 Spring IoC 和 AOP 的理解

IoC

IoC (Inverse of Control:控制反转) 是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由Spring框架来管理。 IoC 在其他语言中也有应用，并非 Spring 特有。 IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map (key, value) ,Map 中存放的是各种对象。

将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

推荐阅读：<https://www.zhihu.com/question/23277575/answer/169698662>

Spring IoC的初始化过程：



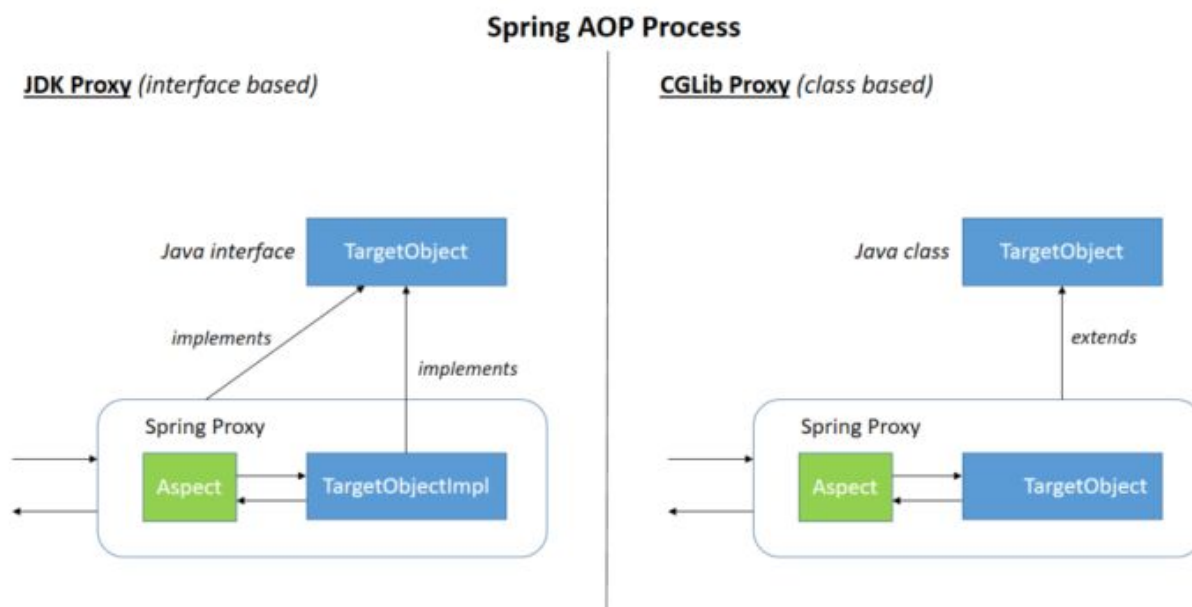
IoC源码阅读

- <https://javadoop.com/post/spring-ioc>

AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK Proxy**，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用**Cglib**，这时候Spring AOP会使用 **Cglib** 生成一个被代理对象的子类来作为代理，如下图所示：



当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ ， AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP。

Spring AOP 和 AspectJ AOP 有什么区别？

Spring AOP 属于运行时增强，而 **AspectJ** 是编译时增强。Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ，AspectJ 应该算的上是 Java 生态系统最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ，它比Spring AOP 快很多。

作者：Guide哥。

介绍: Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

5.1.5 Spring bean

Spring 中的 bean 的作用域有哪些？

- singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session: 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Spring 中的单例 bean 的线程安全问题了解吗？

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。

常见的有两种解决办法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

@Component 和 @Bean 的区别是什么？

1. 作用对象不同：@Component 注解作用于类，而 @Bean 注解作用于方法。
2. @Component 通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（我们可以使用 @ComponentScan 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。@Bean 注解通常是我们标有该注解的方法中定义产生这个 bean，@Bean 告诉了Spring这是某个类的示例，当我需要它的时候还给我。
3. @Bean 注解比 Component 注解的自定义性更强，而且很多地方我们只能通过 @Bean 注解来注册bean。比如当我们引用第三方库中的类需要装配到 Spring 容器时，则只能通过 @Bean 来实现。

@Bean 注解使用示例：

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

上面的代码相当于下面的 xml 配置

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

下面这个例子是通过 @Component 无法实现的。

```

@Bean
public OneService getService(status) {
    case (status) {
        when 1:
            return new serviceImpl1();
        when 2:
            return new serviceImpl2();
        when 3:
            return new serviceImpl3();
    }
}

```

将一个类声明为Spring的 bean 的注解有哪些？

我们一般使用 `@Autowired` 注解自动装配 bean，要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类,采用以下注解可实现：

- `@Component` ：通用的注解，可标注任意类为 Spring 组件。如果一个Bean不知道属于哪个层，可以使用 `@Component` 注解标注。
- `@Repository` ：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service` ：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。
- `@Controller` ：对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

Spring 中的 bean 生命周期？

这部分网上有很多文章都讲到了，下面的内容整理自：<https://yemengying.com/2016/07/14/spring-bean-life-cycle/>，除了这篇文章，再推荐一篇很不错的文章：<https://www.cnblogs.com/zrtqsk/p/3735273.html>。

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个Bean的实例。
- 如果涉及到一些属性值 利用 `set()` 方法设置一些属性值。
- 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法，传入Bean的名字。
- 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoader` 对象的实例。
- 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
- 如果Bean实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()` 方法。
- 如果 Bean 在配置文件中的定义包含 `init-method` 属性，执行指定的方法。

- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessAfterInitialization()` 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 `DisposableBean` 接口，执行 `destroy()` 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 `destroy-method` 属性，执行指定的方法。

图示：

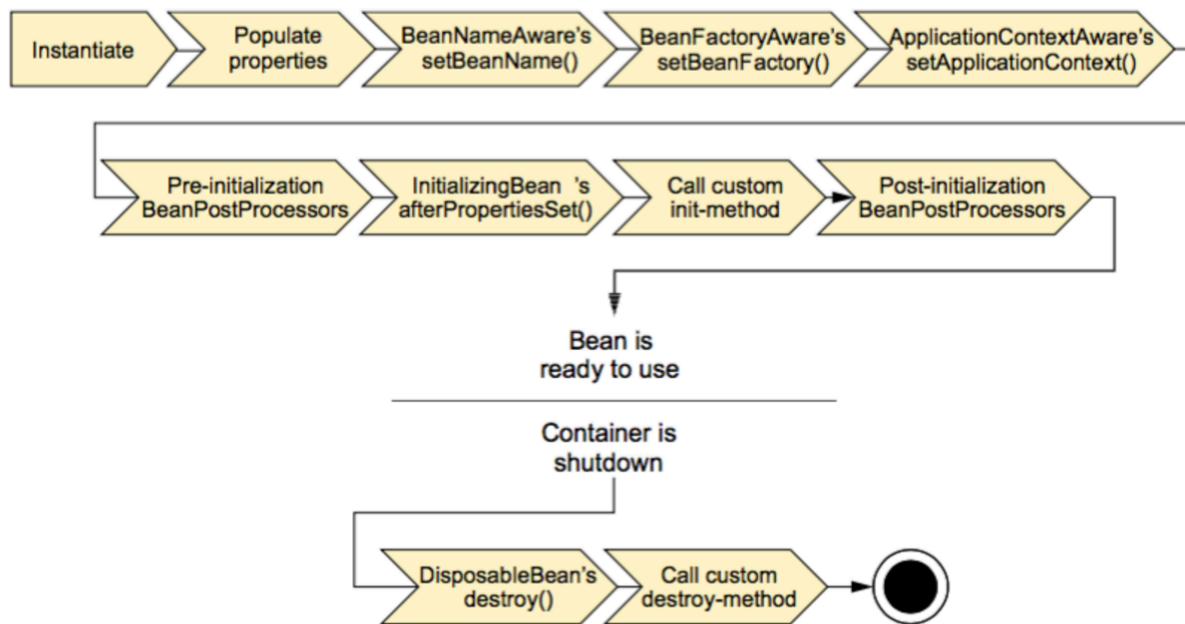
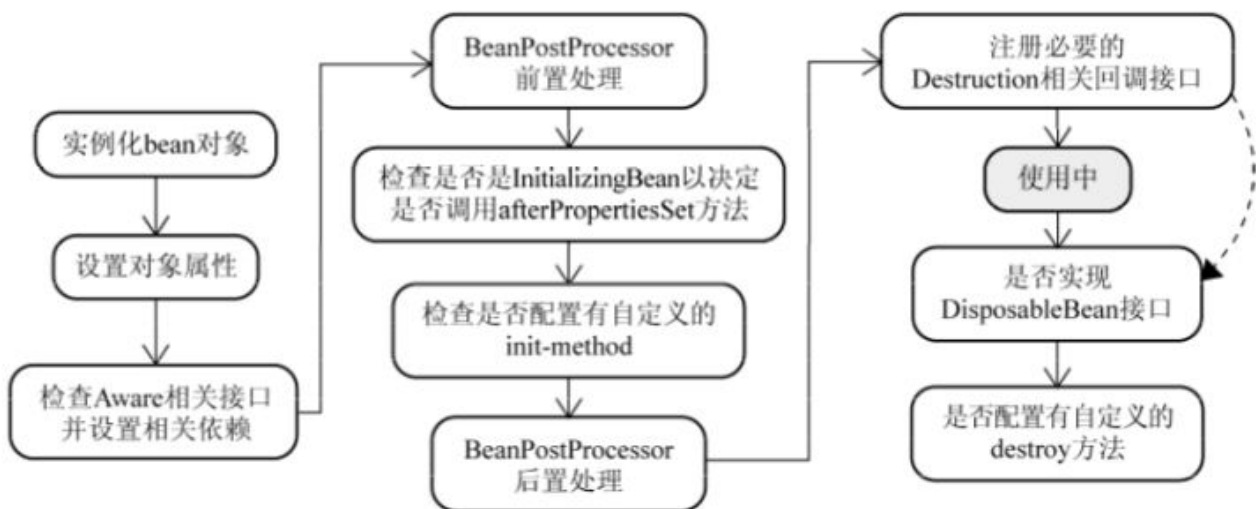


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

与之比较类似的中文版本：



5.1.6 Spring MVC

说说自己对于 Spring MVC 了解？

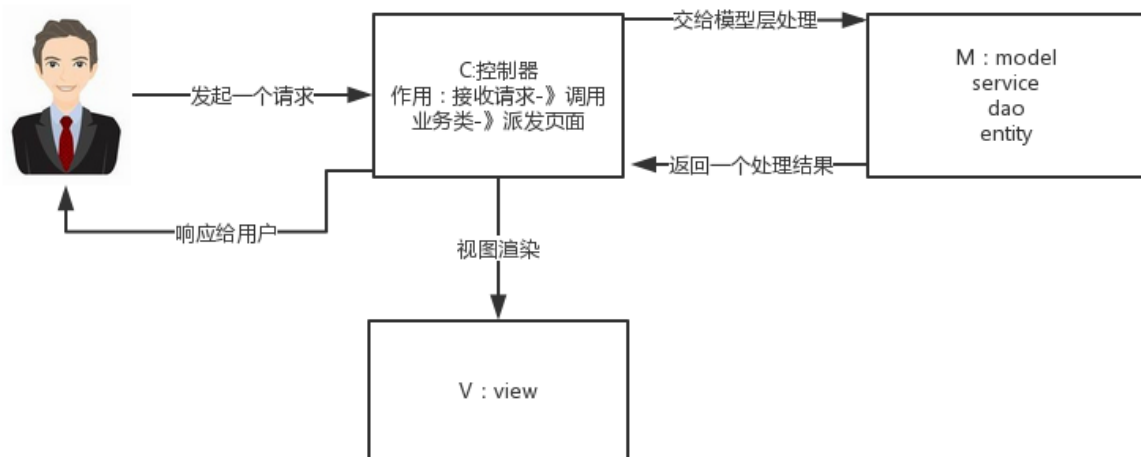
谈到这个问题，我们不得不提之前 Model1 和 Model2 这两个没有 Spring MVC 的时代。

- **Model1 时代**：很多学 Java 后端比较晚的朋友可能并没有接触过 Model1 模式下的 JavaWeb 应用开发。在 Model1 模式下，整个 Web 应用几乎全部用 JSP 页面组成，只用少量的 JavaBean 来处理数据库连接、访问等操作。这个模式下 JSP 即是控制层又是表现层。显而易见，这种模式存在很多问题。比如①将控制逻辑和表现逻辑混杂在一起，导致代码重用率极低；②前端和后端相互依赖，难以进行测试并且开发效率极低；
- **Model2 时代**：学过 Servlet 并做过相关 Demo 的朋友应该了解“Java Bean(Model)+ JSP (View,) +Servlet (Controller)”这种开发模式,这就是早期的 JavaWeb MVC 开发模式。Model:系统涉及的数据，也就是 dao 和 bean。View：展示模型中的数据，只是用来展示。Controller：处理用户请求都发送给，返回数据给 JSP 并展示给用户。

Model2 模式下还存在很多问题，Model2的抽象和封装程度还远远不够，使用Model2进行开发时不可避免地会重复造轮子，这就大大降低了程序的可维护性和复用性。于是很多JavaWeb开发相关的 MVC 框架应运而生比如Struts2，但是 Struts2 比较笨重。随着 Spring 轻量级开发框架的流行，Spring 生态圈出现了 Spring MVC 框架，Spring MVC 是当前最优秀的 MVC 框架。相比于 Struts2，Spring MVC 使用更加简单和方便，开发效率更高，并且 Spring MVC 运行速度更快。

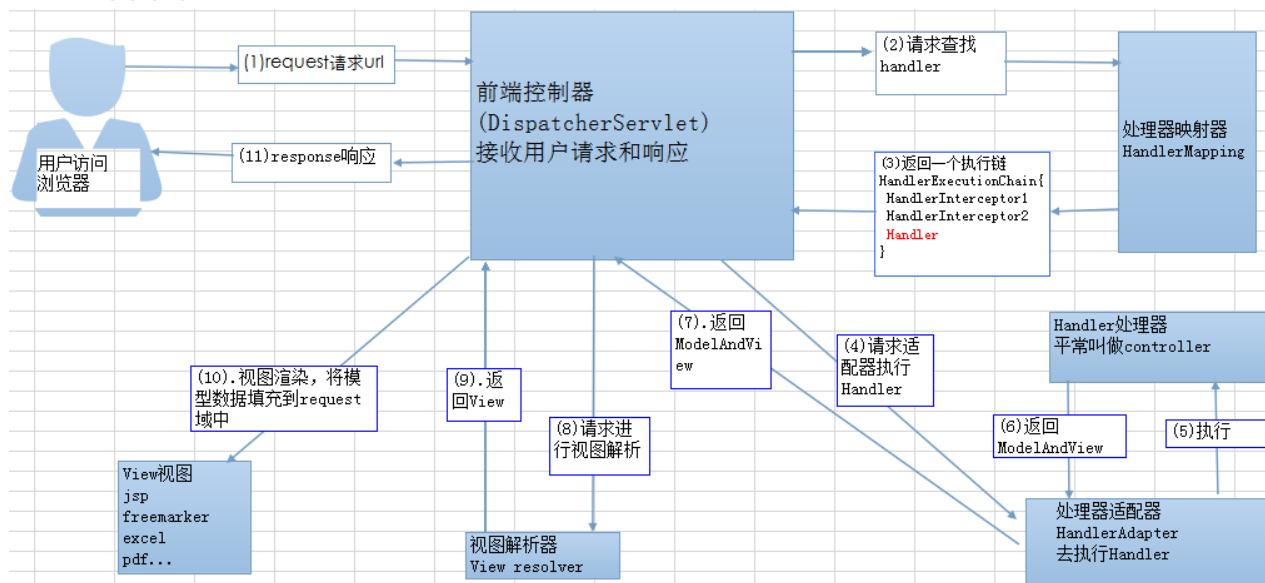
MVC 是一种设计模式, Spring MVC 是一款很优秀的 MVC 框架。Spring MVC 可以帮助我们进行更简洁的Web层的开发，并且它天生与 Spring 框架集成。Spring MVC 下我们一般把后端项目分为 Service层（处理业务）、Dao层（数据库操作）、Entity层（实体类）、Controller层(控制层，返回数据给前台页面)。

Spring MVC 的简单原理图如下：



SpringMVC 工作原理了解吗？

原理如下图所示：



上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 `DispatcherServlet` 的作用是接收请求，响应结果。

流程说明（重要）：

1. 客户端（浏览器）发送请求，直接请求到 `DispatcherServlet`。
2. `DispatcherServlet` 根据请求信息调用 `HandlerMapping`，解析请求对应的 `Handler`。
3. 解析到对应的 `Handler`（也就是我们平常说的 `Controller` 控制器）后，开始由 `HandlerAdapter` 适配器处理。
4. `HandlerAdapter` 会根据 `Handler` 来调用真正的处理器开处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 `ModelAndView` 对象，`Model` 是返回的数据对象，`View` 是个逻辑上的 `View`。
6. `ViewResolver` 会根据逻辑 `View` 查找实际的 `View`。
7. `DispatcherServlet` 把返回的 `Model` 传给 `View`（视图渲染）。
8. 把 `View` 返回给请求者（浏览器）

5.1.7 Spring 框架中用到了哪些设计模式？

关于下面一些设计模式的详细介绍，可以看笔主前段时间的原创文章《面试官：“谈谈Spring中都用了那些设计模式？”》。

- **工厂设计模式**：Spring 使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象。
- **代理设计模式**：Spring AOP 功能的实现。
- **单例设计模式**：Spring 中的 Bean 默认都是单例的。

- **包装器设计模式**：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- **观察者模式**：Spring 事件驱动模型就是观察者模式很经典的一个应用。
- **适配器模式**：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller 。
-

5.1.8 Spring 事务

Spring 管理事务的方式有几种？

1. 编程式事务，在代码中硬编码。(不推荐使用)
2. 声明式事务，在配置文件中配置（推荐使用）

声明式事务又分为两种：

1. 基于XML的声明式事务
2. 基于注解的声明式事务

Spring 事务中的隔离级别有哪几种？

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- **TransactionDefinition.ISOLATION_DEFAULT**: 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED隔离级别。
- **TransactionDefinition.ISOLATION_READ_UNCOMMITTED**: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
- **TransactionDefinition.ISOLATION_READ_COMMITTED**: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- **TransactionDefinition.ISOLATION_REPEATABLE_READ**: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- **TransactionDefinition.ISOLATION_SERIALIZABLE**: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

Spring 事务中哪几种事务传播行为？

支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRED**：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- **TransactionDefinition.PROPAGATION_SUPPORTS**：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- **TransactionDefinition.PROPAGATION_MANDATORY**：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

不支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRES_NEW**：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED**：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NEVER**：以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

- **TransactionDefinition.PROPAGATION_NESTED**：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 **TransactionDefinition.PROPAGATION_REQUIRED**。

@Transactional(rollbackFor = Exception.class)注解了解吗？

我们知道：Exception分为运行时异常RuntimeException和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当 `@Transactional` 注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在 `@Transactional` 注解中如果不配置 `rollbackFor` 属性,那么事物只会在遇到 `RuntimeException` 的时候才会回滚,加上 `rollbackFor=Exception.class` ,可以让事物在遇到非运行时异常时也回滚。

关于 `@Transactional` 注解推荐阅读的文章：

- [透彻的掌握 Spring 中@Transactional 的使用](#)

5.1.9 JPA

如何使用JPA在数据库中非持久化一个字段？

假如我们有下面一个类：

```
Entity(name="USER")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private Long id;

    @Column(name="USER_NAME")
    private String userName;

    @Column(name="PASSWORD")
    private String password;

    private String secret;

}
```

如果我们想让 `secret` 这个字段不被持久化，也就是不被数据库存储怎么办？我们可以采用下面几种方法：

```
static String transient1; // not persistent because of static
final String transient2 = "Satish"; // not persistent because of final
transient String transient3; // not persistent because of transient
@Transient
String transient4; // not persistent because of @Transient
```

一般使用后面两种方式比较多，我个人使用注解的方式比较多。

参考

- 《Spring 技术内幕》
- <http://www.cnblogs.com/wmyskxz/p/8820371.html>
- <https://www.journaldev.com/2696/spring-interview-questions-and-answers>
- <https://www.edureka.co/blog/interview-questions/spring-interview-questions/>
- <https://www.cnblogs.com/clwydjgs/p/9317849.html>
- <https://howtodoinjava.com/interview-questions/top-spring-interview-questions-with-answers/>
- <http://www.tomaszezula.com/2014/02/09/spring-series-part-5-component-vs-bean/>
- <https://stackoverflow.com/questions/34172888/difference-between-bean-and-autowired>

公众号

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《Java面试突击》：由本文档衍生的专为面试而生的《Java面试突击》V2.0 PDF 版本[公众号](#)后台回复 "Java面试突击" 即可免费领取！

Java工程师必备学习资源：一些Java工程师常用学习资源公众号后台回复关键字“1”即可免费无套路获取。



5.2 MyBatis面试题总结

本篇文章是JavaGuide收集自网络，原出处不明。

Mybatis 技术内幕系列博客，从原理和源码角度，介绍了其内部实现细节，无论是写的好与不好，我确实是用心得写了，由于并不是介绍如何使用 Mybatis 的文章，所以，一些参数使用细节略掉了，我们的目标是介绍 Mybatis 的技术架构和重要组成部分，以及基本运行原理。

博客写的很辛苦，但是写出来却不一定好看，所谓开始很兴奋，过程很痛苦，结束很遗憾。要求不高，只要读者能从系列博客中，学习到一点其他博客所没有的技术点，作为作者，我就很欣慰了，我也读别人写的博客，通常对自己当前研究的技术，是很有帮助的。

尽管还有很多可写的内容，但是，我认为再写下去已经没有意义，任何其他小的功能点，都是在已经介绍的基本框架和基本原理下运行的，只有结束，才能有新的开始。写博客也积攒了一些经验，源码多了感觉就是复制黏贴，源码少了又觉得是空谈原理，将来再写博客，我希望能是“精炼博文”，好读易懂美观读起来又不累，希望自己能再写一部开源分布式框架原理系列博客。

有胆就来，我出几道 Mybatis 面试题，看你能回答上来几道（都是我出的，可不是网上找的）。

5.2.1 #{}和\${}的区别是什么？

注：这道题是面试官面试我同事的。

答：

- `${}` 是 Properties 文件中的变量占位符，它可以用于标签属性值和 sql 内部，属于静态文本替换，比如 `${driver}` 会被静态替换为 `com.mysql.jdbc.Driver`。
- `#{}` 是 sql 的参数占位符，Mybatis 会将 sql 中的 `#{}` 替换为 `?` 号，在 sql 执行前会使用 `PreparedStatement` 的参数设置方法，按序给 sql 的 `?` 号占位符设置参数值，比如 `ps.setInt(0, parameterValue)`，`#{}item.name` 的取值方式为使用反射从参数对象中获取 item 对象的 name 属性值，相当于 `param.getItem().getName()`。

5.2.2 Xml 映射文件中，除了常见的 `select`、`insert`、`update`、`delete` 标签之外，还有哪些标签？

注：这道题是京东面试官面试我时间的。

答：还有很多其他的标

签，`<resultMap>`、`<parameterMap>`、`<sql>`、`<include>`、`<selectKey>`，加上动态 sql 的 9 个标签，`trim`、`where`、`set`、`foreach`、`if`、`choose`、`when`、`otherwise`、`bind` 等，其中为 sql 片段标签，通过 `<include>` 标签引入 sql 片段，`<selectKey>` 为不支持自增的主键生成策略标签。

5.2.3 最佳实践中，通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应，请问，这个 Dao 接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载吗？

注：这道题也是京东面试官面试我时间的。

答：Dao 接口，就是人们常说的 Mapper 接口，接口的全限定名，就是映射文件中的 namespace 的值，接口的方法名，就是映射文件中 MappedStatement 的 id 值，接口方法内的参数，就是传递给 sql 的参数。Mapper 接口是没有实现类的，当调用接口方法时，接口全限定名+方法名拼接字符串作为 key 值，可唯一定位一个 MappedStatement，举

例：`com.mybatis3.mappers.StudentDao.findStudentById`，可以唯一找到 namespace 为 `com.mybatis3.mappers.StudentDao` 下面 `id = findStudentById` 的 MappedStatement。在 Mybatis 中，每一个 `<select>`、`<insert>`、`<update>`、`<delete>` 标签，都会被解析为一个 MappedStatement 对象。

Dao 接口里的方法，是不能重载的，因为是全限定名+方法名的保存和寻找策略。

Dao 接口的工作原理是 JDK 动态代理，Mybatis 运行时会使用 JDK 动态代理为 Dao 接口生成代理 proxy 对象，代理对象 proxy 会拦截接口方法，转而执行 MappedStatement 所代表的 sql，然后将 sql 执行结果返回。

5.2.4 Mybatis 是如何进行分页的？分页插件的原理是什么？

注：我出的。

答：Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页，可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

举例：`select _ from student`，拦截 sql 后重写为：`select t._ from (select * from student) t limit 0, 10`

5.2.5 简述 Mybatis 的插件运行原理，以及如何编写一个插件。

注：我出的。

答：Mybatis 仅可以编写针对

`ParameterHandler`、`ResultSetHandler`、`StatementHandler`、`Executor` 这 4 种接口的插件，Mybatis 使用 JDK 的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 `InvocationHandler` 的 `invoke()` 方

法，当然，只会拦截那些你指定需要拦截的方法。

实现 Mybatis 的 Interceptor 接口并复写 `intercept()` 方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

5.2.6 Mybatis 执行批量插入，能返回数据库主键列表吗？

注：我出的。

答：能，JDBC 都能，Mybatis 当然也能。

5.2.7 Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？

注：我出的。

答：Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能，Mybatis 提供了 9 种动态 sql 标签

`trim|where|set|foreach|if|choose|when|otherwise|bind`。

其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

5.2.8 Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？

注：我出的。

答：第一种是使用 `<resultMap>` 标签，逐一指定列名和对象属性名之间的映射关系。第二种是使用 sql 列的别名功能，将列别名书写为对象属性名，比如 `T_NAME AS NAME`，对象属性名一般是 `name`，小写，但是列名不区分大小写，Mybatis 会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成 `T_NAME AS NaMe`，Mybatis 一样可以正常工作。

有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

5.2.9 Mybatis 能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别。

注：我出的。

答：能，Mybatis 不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把 `selectOne()` 修改为 `selectList()` 即可；多对多查询，其实就是一对多查询，只需要把 `selectOne()` 修改为 `selectList()` 即可。

关联对象查询，有两种实现方式，一种是单独发送一个 sql 去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用 join 查询，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。

那么问题来了，join 查询出来 100 条记录，如何确定主对象是 5 个，而不是 100 个？其去重复的原理是 `<resultMap>` 标签内的 `<id>` 子标签，指定了唯一确定一条记录的 id 列，Mybatis 根据列值来完成 100 条记录的去重复功能，`<id>` 可以有多个，代表了联合主键的语意。

同样主对象的关联对象，也是根据这个原理去重复的，尽管一般情况下，只有主对象会有重复记录，关联对象一般不会重复。

举例：下面 join 查询出来 6 条记录，一、二列是 Teacher 对象列，第三列为 Student 对象列，Mybatis 去重复处理后，结果为 1 个老师 6 个学生，而不是 6 个老师 6 个学生。

```
t_id t_name s_id
```

```
| 1 | teacher | 38 |
| 1 | teacher | 39 |
| 1 | teacher | 40 |
| 1 | teacher | 41 |
| 1 | teacher | 42 |
| 1 | teacher | 43 |
```

5.2.10 Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

注：我出的。

答：Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 `lazyLoadingEnabled=true/false`。

它的原理是，使用 `CGLIB` 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 `a.getB().getName()`，拦截器 `invoke()` 方法发现 `a.getB()` 是 `null` 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 `a.setB(b)`，于是 a 的对象 b 属性就有值了，接着完成 `a.getB().getName()` 方法的调用。这就是延迟加载的基本原理。

当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。

5.2.11 Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

注：我出的。

答：不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；毕竟 namespace 不是必须的，只是最佳实践而已。

原因就是 namespace+id 是作为 `Map<String, MappedStatement>` 的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

5.2.12 Mybatis 中如何执行批处理？

注：我出的。

答：使用 `BatchExecutor` 完成批处理。

5.2.13 Mybatis 都有哪些 Executor 执行器？它们之间的区别是什么？

注：我出的

答：Mybatis 有三种基本的 Executor 执行器，`SimpleExecutor`、`ReuseExecutor`、`BatchExecutor`。

SimpleExecutor：每执行一次 update 或 select，就开启一个 Statement 对象，用完立刻关闭 Statement 对象。

ReuseExecutor：执行 update 或 select，以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完，不关闭 Statement 对象，而是放置于 `Map<String, Statement>` 内，供下一次使用。简言之，就是重复使用 Statement 对象。

BatchExecutor：执行 update（没有 select，JDBC 批处理不支持 select），将所有 sql 都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement 对象，每个 Statement 对象都是 addBatch() 完毕后，等待逐一执行 executeBatch() 批处理。与 JDBC 批处理相同。

作用范围：Executor 的这些特点，都严格限制在 SqlSession 生命周期范围内。

5.2.14 Mybatis 中如何指定使用哪一种 Executor 执行器？

注：我出的

答：在 Mybatis 配置文件中，可以指定默认的 ExecutorType 执行器类型，也可以手动给 DefaultSqlSessionFactory 的创建 SqlSession 的方法传递 ExecutorType 类型参数。

5.2.15 Mybatis 是否可以映射 Enum 枚举类？

注：我出的

答：Mybatis 可以映射枚举类，不单可以映射枚举类，Mybatis 可以映射任何对象到表的一列上。映射方式为自定义一个 TypeHandler，实现 TypeHandler 的 setParameter() 和 getResult() 接口方法。TypeHandler 有两个作用，一是完成从 javaType 至 jdbcType 的转换，二是完成 jdbcType 至 javaType 的转换，体现为 setParameter() 和 getResult() 两个方法，分别代表设置 sql 问号占位符参数和获取列查询结果。

5.2.16 Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能否定义在 A 标签的后面，还是说必须定义在 A 标签的前面？

注：我出的

答：虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。

原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不存在，此时，Mybatis 会将 A 标签标记为未解析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也就可以正常解析完成了。

5.2.17 简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系？

注：我出的

答：Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中，`<parameterMap>` 标签会被解析为 `ParameterMap` 对象，其每个子元素会被解析为 `ParameterMapping` 对象。`<resultMap>` 标签会被解析为 `ResultMap` 对象，其每个子元素会被解析为 `ResultMapping` 对象。每一个 `<select>`、`<insert>`、`<update>`、`<delete>` 标签均会被解析为 `MappedStatement` 对象，标签内的 sql 会被解析为 `BoundSql` 对象。

5.2.18 为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

注：我出的

答：Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

面试题看似都很简单，但是想要能正确回答上来，必定是研究过源码且深入的人，而不是仅会使用的人或者用的很熟的人，以上所有面试题及其答案所涉及的内容，在我的 Mybatis 系列博客中都有详细讲解和原理分析。-----

5.3 Kafka面试题总结

5.3.1 Kafka 是什么？主要应用场景有哪些？

Kafka 是一个分布式流式处理平台。这到底是什么意思呢？

流平台具有三个关键功能：

1. **消息队列**：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
2. **容错的持久方式存储记录消息流**：Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险。
3. **流式处理平台**：在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

Kafka 主要有两大应用场景：

1. **消息队列**：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
2. **数据处理**：构建实时的流数据处理程序来转换或处理数据流。

5.3.2 和其他消息队列相比,Kafka的优势在哪里？

我们现在经常提到 Kafka 的时候就已经默认它是一个非常优秀的消息队列了，我们也会经常拿它给 RocketMQ、RabbitMQ 对比。我觉得 Kafka 相比其他消息队列主要的优势如下：

1. **极致的性能**：基于 Scala 和 Java 语言开发，设计中大量使用了批量处理和异步的思想，最高可以每秒处理千万级别的消息。
2. **生态系统兼容性无可匹敌**：Kafka 与周边生态系统的兼容性是最好的没有之一，尤其在大数据和流计算领域。

实际上在早期的时候 Kafka 并不是一个合格的消息队列，早期的 Kafka 在消息队列领域就像是一个衣衫褴褛的孩子一样，功能不完备并且有一些小问题比如丢失消息、不保证消息可靠性等等。当然，这也和 LinkedIn 最早开发 Kafka 用于处理海量的日志有很大关系，哈哈，人家本来最开始就不是为了作为消息队列滴，谁知道后面误打误撞在消息队列领域占据了一席之地。

随着后续的发展，这些短板都被 Kafka 逐步修复完善。所以，**Kafka 作为消息队列不可靠**这个说法已经过时！

5.3.3 队列模型了解吗？Kafka 的消息模型知道吗？

题外话：早期的 JMS 和 AMQP 属于消息服务领域权威组织所做的相关的标准，我在 [JavaGuide](#) 的《[消息队列其实很简单](#)》这篇文章中介绍过。但是，这些标准的进化跟不上消息队列的演进速度，这些标准实际上已经属于废弃状态。所以，可能存在的情况是：不同的消息队列都有自己的一套消息模型。

队列模型：早期的消息模型

队列模型

作者：SnailClimb
公众号&Github：JavaGuide



使用队列（Queue）作为消息通信载体，满足生产者与消费者模式，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。比如：我们生产者发送 100 条消息的话，两个消费者来消费一般情况下两个消费者会按照消息发送的顺序各自消费一半（也就是你一个我一个的消费。）

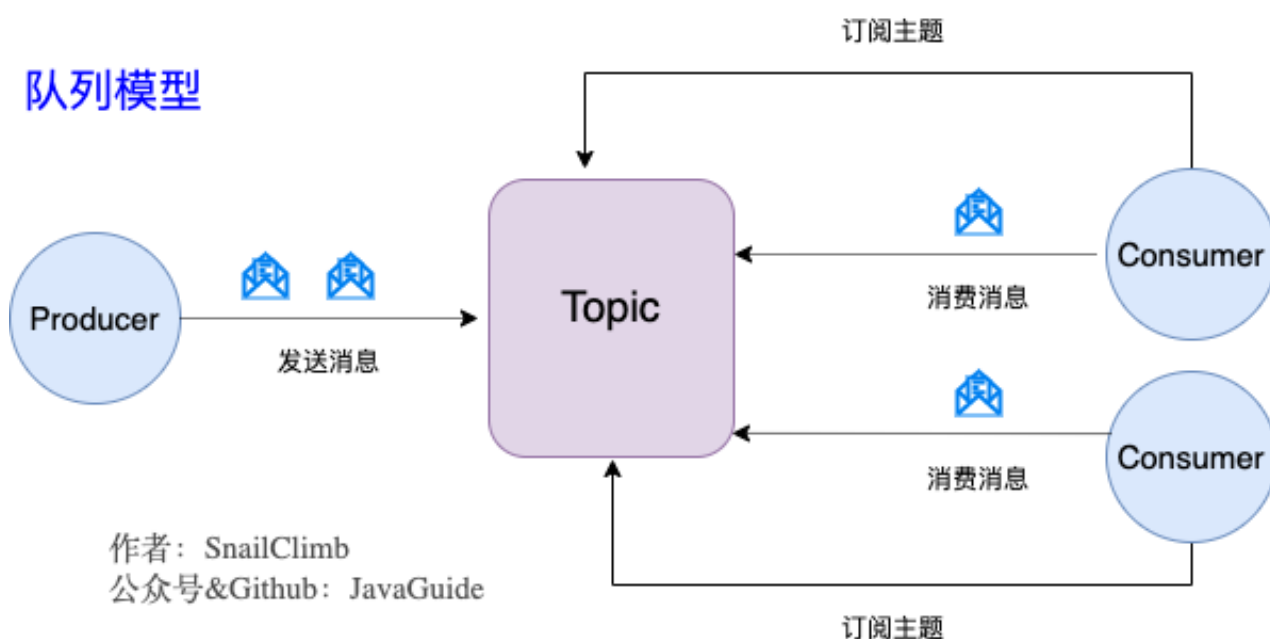
队列模型存在的问题：

假如我们存在这样一种情况：我们需要将生产者产生的消息分发给多个消费者，并且每个消费者都能接收到完整的消息内容。

这种情况，队列模型就不好解决了。很多比较杠精的人就说：我们可以为每个消费者创建一个单独的队列，让生产者发送多份。这是一种非常愚蠢的做法，浪费资源不说，还违背了使用消息队列的目的。

发布-订阅模型:Kafka 消息模型

发布-订阅模型主要是为了解决队列模型存在的问题。



发布订阅模型（Pub-Sub）使用主题（Topic）作为消息通信载体，类似于广播模式；发布者发布一条消息，该消息通过主题传递给所有的订阅者，在一条消息广播之后才订阅的用户则是收不到该条消息的。

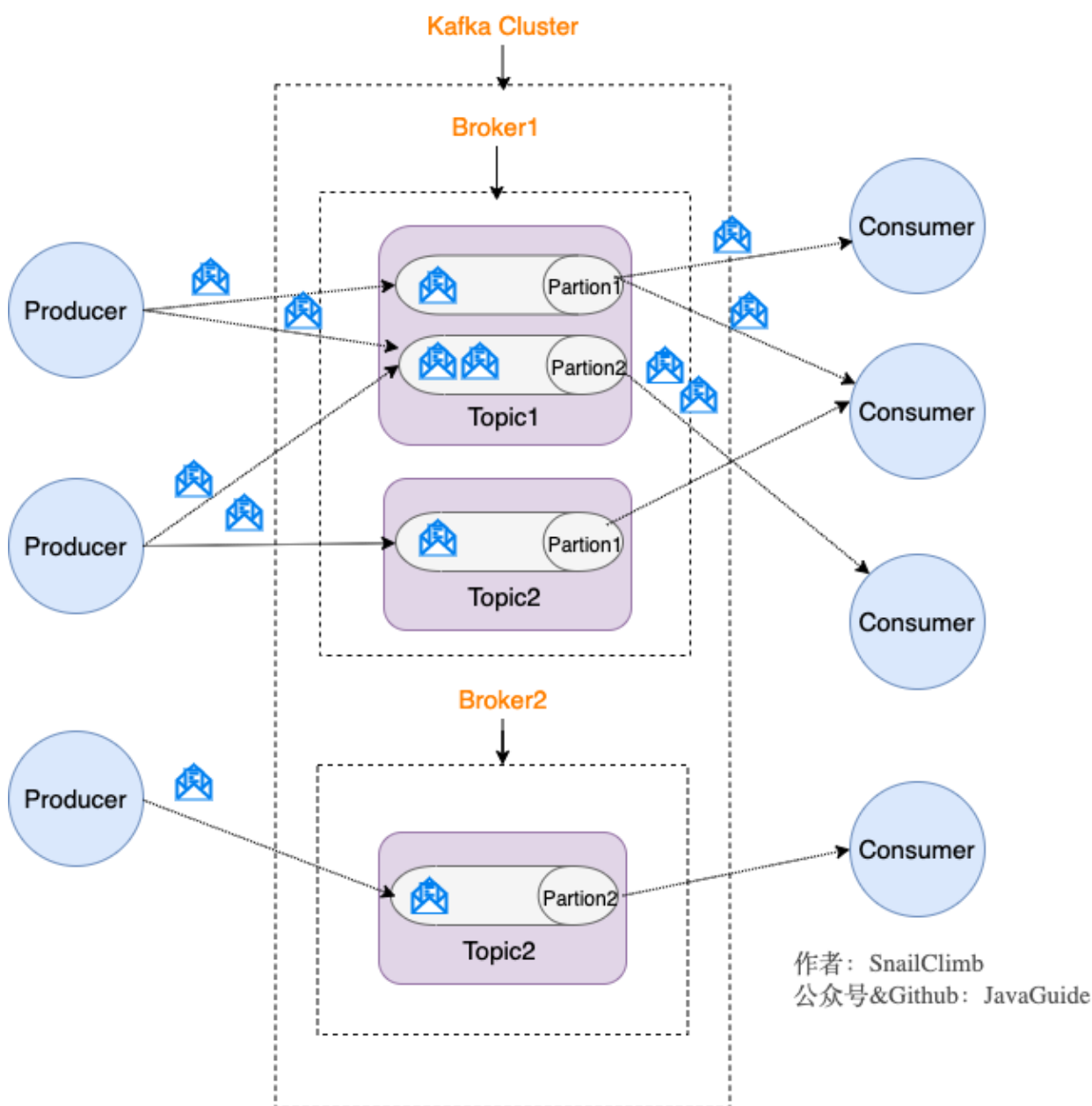
在发布 - 订阅模型中，如果只有一个订阅者，那它和队列模型就基本是一样的了。所以说，发布 - 订阅模型在功能层面上是可以兼容队列模型的。

Kafka 采用的就是发布 - 订阅模型。

RocketMQ 的消息模型和 Kafka 基本是完全一样的。唯一的区别是 Kafka 中没有队列这个概念，与之对应的是 Partition（分区）。

5.3.4 什么是Producer、Consumer、Broker、Topic、Partition?

Kafka 将生产者发布的消息发送到 **Topic（主题）** 中，需要这些消息的消费者可以订阅这些 **Topic（主题）**，如下图所示：



上面这张图也为我们引出了，Kafka 比较重要的几个概念：

1. **Producer（生产者）**：产生消息的一方。
2. **Consumer（消费者）**：消费消息的一方。

3. **Broker (代理)** : 可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。

同时, 你一定也注意到每个 Broker 中又包含了 Topic 以及 Partition 这两个重要的概念:

- **Topic (主题)** : Producer 将消息发送到特定的主题, Consumer 通过订阅特定的 Topic(主题) 来消费消息。
- **Partition (分区)** : Partition 属于 Topic 的一部分。一个 Topic 可以有多个 Partition , 并且同一 Topic 下的 Partition 可以分布在不同的 Broker 上, 这也就表明一个 Topic 可以横跨多个 Broker 。这正如我上面所画的图一样。

划重点: **Kafka 中的 Partition (分区)** 实际上可以对应成为消息队列中的队列。这样是不是更好理解一点?

5.3.5 Kafka 的多副本机制了解吗? 带来了什么好处?

还有一点我觉得比较重要的是 Kafka 为分区 (Partition) 引入了多副本 (Replica) 机制。分区 (Partition) 中的多个副本之间会有一个叫做 leader 的家伙, 其他副本称为 follower。我们发送的消息会被发送到 leader 副本, 然后 follower 副本才能从 leader 副本中拉取消息进行同步。

生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝, 它们的存在只是为了保证消息存储的安全性。当 leader 副本发生故障时会从 follower 中选举出一个 leader, 但是 follower 中如果有和 leader 同步程度达不到要求的参加不了 leader 的竞选。

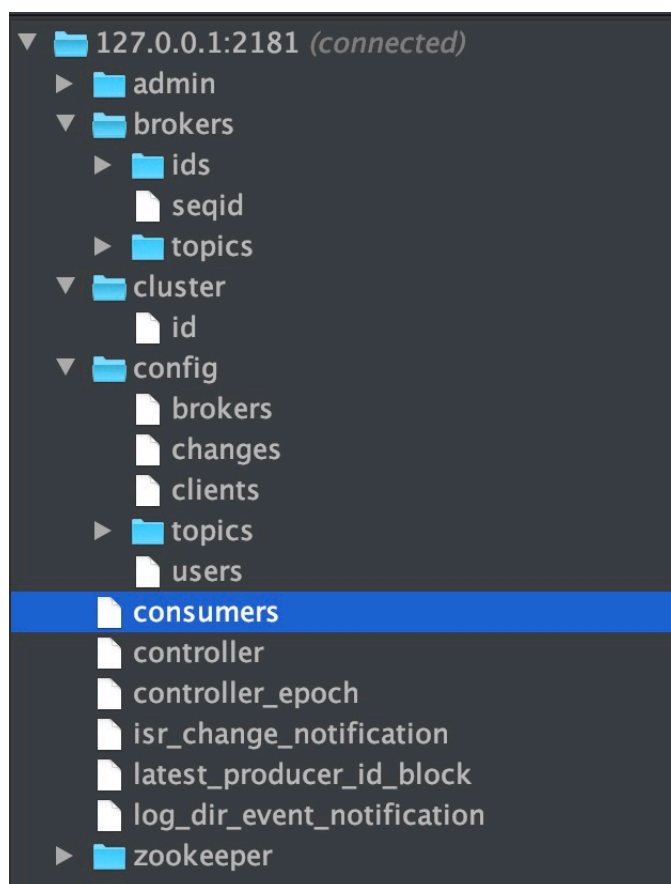
Kafka 的多分区 (Partition) 以及多副本 (Replica) 机制有什么好处呢?

1. Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力 (负载均衡)。
2. Partition 可以指定对应的 Replica 数, 这也极大地提高了消息存储的安全性, 提高了容灾能力, 不过也相应的增加了所需要的存储空间。

5.3.6 Zookeeper 在 Kafka 中的作用知道吗?

要想搞懂 zookeeper 在 Kafka 中的作用 一定要自己搭建一个 Kafka 环境然后自己进 zookeeper 去看一下有哪些文件夹和 Kafka 有关, 每个节点又保存了什么信息。一定不要光看不实践, 这样学来的也终会忘记! 这部分内容参考和借鉴了这篇文章: <https://www.jianshu.com/p/a036405f989c>。

下图就是我的本地 Zookeeper，它成功和我本地的 Kafka 关联上（以下文件夹结构借助 idea 插件 Zookeeper tool 实现）。



ZooKeeper 主要为 Kafka 提供元数据的管理的功能。

从图中我们可以看出，Zookeeper 主要为 Kafka 做了下面这些事情：

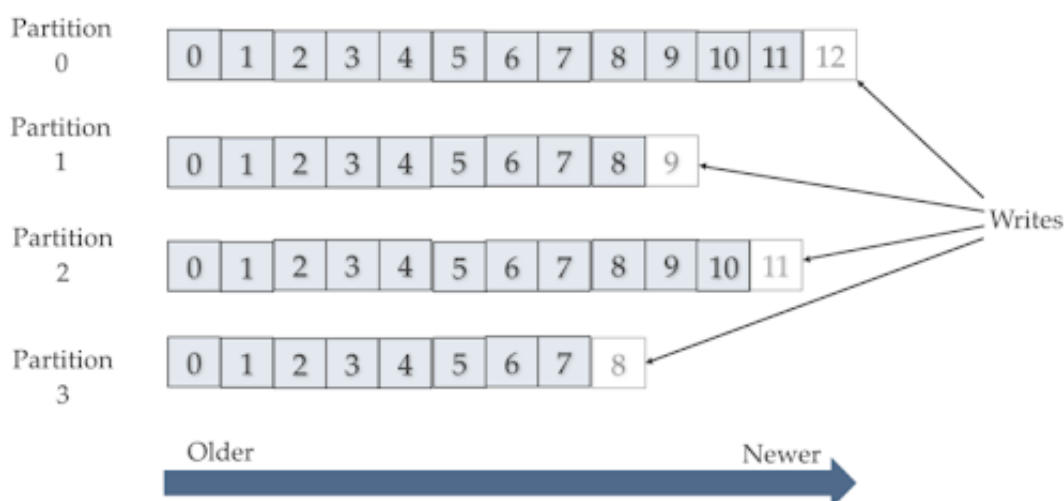
1. **Broker 注册**：在 Zookeeper 上会有一个专门用来进行 **Broker** 服务器列表记录的节点。每个 Broker 在启动时，都会到 Zookeeper 上进行注册，即到/brokers/ids 下创建属于自己的节点。每个 Broker 就会将自己的 IP 地址和端口等信息记录到该节点中去
2. **Topic 注册**：在 Kafka 中，同一个**Topic** 的消息会被分成多个分区并将其分布在多个 Broker 上，这些分区信息及与 **Broker** 的对应关系也都是由 Zookeeper 在维护。比如我创建了一个名字为 my-topic 的主题并且它有两个分区，对应到 zookeeper 中会创建这些文件夹： /brokers/topics/my-topic/Partitions/0 、 /brokers/topics/my-topic/Partitions/1
3. **负载均衡**：上面也说过 Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力。对于同一个 Topic 的不同 Partition, Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生消息后也会尽量投递到不同 Broker 的 Partition 里面。当 Consumer 消费的时候，Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数量来实现动态负载均衡。
4.

5.3.7 Kafka 如何保证消息的消费顺序？

我们在使用消息队列的过程中经常有业务场景需要严格保证消息的消费顺序，比如我们同时发了 2 个消息，这 2 个消息对应的操作分别对应的数据库操作是：更改用户会员等级、根据会员等级计算订单价格。假如这两条消息的消费顺序不一样造成的最终结果就会截然不同。

我们知道 Kafka 中 Partition(分区)是真正保存消息的地方，我们发送的消息都被放在了这里。而我们的 Partition(分区) 又存在于 Topic(主题) 这个概念中，并且我们可以给特定 Topic 指定多个 Partition。

Kafka Topic Partitions Layout



每次添加消息到 Partition(分区) 的时候都会采用尾加法，如上图所示。Kafka 只能为我们保证 Partition(分区) 中的消息有序，而不能保证 Topic(主题) 中的 Partition(分区) 的有序。

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 (offset)。Kafka 通过偏移量 (offset) 来保证消息在分区内的顺序性。

所以，我们就有一种很简单的保证消息消费顺序的方法：**1 个 Topic 只对应一个 Partition**。这样当然可以解决问题，但是破坏了 Kafka 的设计初衷。

Kafka 中发送 1 条消息的时候，可以指定 topic, partition, key,data (数据) 4 个参数。如果你发送消息的时候指定了 Partition 的话，所有消息都会被发送到指定的 Partition。并且，同一个 key 的消息可以保证只发送到同一个 partition，这个我们可以采用表/对象的 id 来作为 key。

总结一下，对于如何保证 Kafka 中消息消费的顺序，有了下面两种方法：

1. 1 个 Topic 只对应一个 Partition。
2. （推荐）发送消息的时候指定 key/Partition。

当然不仅仅只有上面两种方法，上面两种方法是我觉得比较好理解的，

5.3.8 Kafka 如何保证消息不丢失

生产者丢失消息的情况

生产者(Producer) 调用 `send` 方法发送消息之后，消息可能因为网络问题并没有发送过去。

所以，我们不能默认在调用 `send` 方法发送消息之后消息发送成功了。为了确定消息是发送成功，我们要判断消息发送的结果。但是要注意的是 Kafka 生产者(Producer) 使用 `send` 方法发送消息实际上是异步的操作，我们可以通过 `get()` 方法获取调用结果，但是这样也让它变为了同步操作，示例代码如下：

详细代码见我的这篇文章：[Kafka系列第三篇！10 分钟学会如何在 Spring Boot 程序中使用 Kafka 作为消息队列？](#)

```
SendResult<String, Object> sendResult = kafkaTemplate.send(topic, o).get();
if (sendResult.getRecordMetadata() != null) {
    logger.info("生产者成功发送消息到" + sendResult.getProducerRecord().topic() +
        "-> " + sendRe
            sult.getProducerRecord().value().toString());
}
```

但是一般不推荐这么做！可以采用为其添加回调函数的形式，示例代码如下：

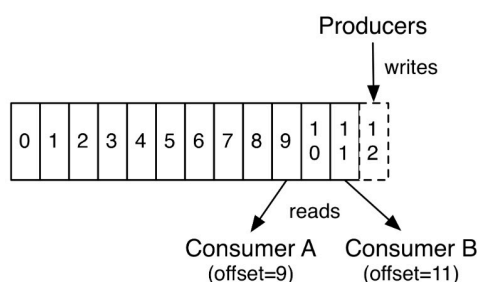
```
ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, o);
future.addCallback(result -> logger.info("生产者成功发送消息到topic:{}
partition:{}的消息", result.getRecordMetadata().topic(),
result.getRecordMetadata().partition()),
    ex -> logger.error("生产者发送失败，原因：{}",
ex.getMessage()));
```

如果消息发送失败的话，我们检查失败的原因之后重新发送即可！

另外这里推荐为 **Producer** 的 `retries`（重试次数）设置一个比较合理的值，一般是 3，但是为了保证消息不丢失的话一般会设置比较大一点。设置完成之后，当出现网络问题之后能够自动重试消息发送，避免消息丢失。另外，建议还要设置重试间隔，因为间隔太小的话重试的效果就不明显了，网络波动一次你3次一下子就重试完了

消费者丢失消息的情况

我们知道消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量（offset）。偏移量（offset）表示 Consumer 当前消费到的 Partition(分区)的所在的位置。Kafka 通过偏移量（offset）可以保证消息在分区内的顺序性。



In fact, the only metadata retained on a per-consumer basis is the offset or position of that consumer in the log. This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes. For example a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".

当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提交的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费的时候，突然挂掉了，消息实际上并没有被消费，但是 offset 却被自动提交了。

解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息之后之后再自己手动提交 offset。但是，细心的朋友一定会发现，这样会带来消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果自己挂掉了，那么这个消息理论上就会被消费两次。

Kafka 弄丢了消息

我们知道 Kafka 为分区（Partition）引入了多副本（Replica）机制。分区（Partition）中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。

试想一种情况：假如 leader 副本所在的 broker 突然挂掉，那么就要从 follower 副本重新选出一个 leader，但是 leader 的数据还有一些没有被 follower 副本的同步的话，就会造成消息丢失。

设置 `acks = all`

解决办法就是我们设置 `acks = all`。`acks` 是 Kafka 生产者(Producer) 很重要的一个参数。

`acks` 的默认值即为1，代表我们的消息被leader副本接收之后就算被成功发送。当我们配置 `acks = all` 代表则所有副本都要接收到该消息之后该消息才算真正成功被发送。

设置 `replication.factor >= 3`

为了保证 leader 副本能有 follower 副本能同步消息，我们一般会为 topic 设置 `replication.factor >= 3`。这样就可以保证每个 分区(partition) 至少有 3 个副本。虽然造成了数据冗余，但是带来了数据的安全性。

设置 `min.insync.replicas > 1`

一般情况下我们还需要设置 `min.insync.replicas > 1`，这样配置代表消息至少要被写入到 2 个副本才算是被成功发送。`min.insync.replicas` 的默认值为 1，在实际生产中应尽量避免默认值 1。

但是，为了保证整个 Kafka 服务的高可用性，你需要确保 `replication.factor > min.insync.replicas`。为什么呢？设想一下加入两者相等的话，只要是有一个副本挂掉，整个分区就无法正常工作了。这明显违反高可用性！一般推荐设置成 `replication.factor = min.insync.replicas + 1`。

设置 `unclean.leader.election.enable = false`

Kafka 0.11.0.0版本开始 `unclean.leader.election.enable` 参数的默认值由原来的true 改为 false

我们最开始也说了我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。多个 follower 副本之间的消息同步情况不一样，当我们配置了 `unclean.leader.election.enable = false` 的话，当 leader 副本发生故障时就不会从 follower 副本中和 leader 同步程度达不到要求的副本中选择出 leader，这样降低了消息丢失的可能性。

5.3.9 Kafka 如何保证消息不重复消费

代办...

Reference

- Kafka 官方文档: <https://kafka.apache.org/documentation/>
- 极客时间 — 《Kafka核心技术与实战》第11节: 无消息丢失配置怎么实现?

5.4 Netty 面试题总结

Netty 总算总结完了, Guide 也是长舒了一口气。有太多读者私信我让我总结 Netty 了, 因为经常会在面试中碰到 Netty 相关的问题。

全文采用大家喜欢的与面试官对话的形式展开。如果大家觉得 Guide 总结的不错的话, 不妨向好朋友们推荐一下 JavaGuide, 这是最好礼物, 哈哈!

5.4.1 Netty 是什么?

 **面试官**: 介绍一下自己对 Netty 的认识吧! 小伙子。

 **我**: 好的! 那我就简单用 3 点来概括一下 Netty 吧!

1. Netty 是一个 **基于 NIO** 的 client-server(客户端服务器)框架, 使用它可以快速简单地开发网络应用程序。
2. 它极大地简化并优化了 TCP 和 UDP 套接字服务器等网络编程, 并且性能以及安全性等很多方面甚至都要更好。
3. **支持多种协议** 如 FTP, SMTP, HTTP 以及各种二进制和基于文本的传统协议。


用官方的总结就是: **Netty 成功地找到了一种在不妥协可维护性和性能的情况下实现易于开发, 性能, 稳定性和灵活性的方法。**

除了上面介绍的之外, 很多开源项目比如我们常用的 Dubbo、RocketMQ、Elasticsearch、gRPC 等等都用到了 Netty。

网络编程我愿意称中 Netty 为王。

5.4.2 为什么要用 Netty?

 **面试官**: 为什么要用 Netty 呢? 能不能说一下自己的看法。


 **我**: 因为 Netty 具有下面这些优点, 并且相比于直接使用 JDK 自带的 NIO 相关的 API 来说更加易用。

- 统一的 API, 支持多种传输类型, 阻塞和非阻塞的。

- 简单而强大的线程模型。
- 自带编解码器解决 TCP 粘包/拆包问题。
- 自带各种协议栈。
- 真正的无连接数据包套接字支持。
- 比直接使用 Java 核心 API 有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
- 安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。
- 社区活跃
- 成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty，比如我们经常接触的 Dubbo、RocketMQ 等等。
-

5.4.3 Netty 应用场景了解么？


 **面试官**：能不能通俗地说一下使用 Netty 可以做什么事情？

 **我**：凭借自己的了解，简单说一下吧！理论上来说，NIO 可以做的事情，使用 Netty 都可以做并且更好。Netty 主要用来做**网络通信**：

1. **作为 RPC 框架的网络通信工具**：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务节点之间的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！
2. **实现一个自己的 HTTP 服务器**：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为 Java 后端开发，我们一般使用 Tomcat 比较多。一个最基本的 HTTP 服务器可要以处理常见的 HTTP Method 的请求，比如 POST 请求、GET 请求等等。
3. **实现一个即时通讯系统**：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。
4. ****实现消息推送系统****：市面上有很多消息推送系统都是基于 Netty 来做的。
5.

5.4.4 Netty 核心组件有哪些？分别有什么作用？

 **面试官**：Netty 核心组件有哪些？分别有什么作用？

 **我**：表面上，嘴上开始说起 Netty 的核心组件有哪些，实则，内心已经开始 mmp 了，深度怀疑这面试官是存心搞我啊！

1.Channel

`Channel` 接口是 Netty 对网络操作抽象类，它除了包括基本的 I/O 操作，如 `bind()`、`connect()`、`read()`、`write()` 等。

比较常用的 `Channel` 接口实现类是 `NioServerSocketChannel`（服务端）和 `NioSocketChannel`（客户端），这两个 `Channel` 可以和 BIO 编程模型中的 `ServerSocket` 以及 `Socket` 两个概念对应上。Netty 的 `Channel` 接口所提供的 API，大大地降低了直接使用 `Socket` 类的复杂性。

2.EventLoop

这么说吧！`EventLoop`（事件循环）接口可以说是 Netty 中最核心的概念了！

《Netty 实战》这本书是这样介绍它的：

`EventLoop` 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。

是不是很难理解？说实话，我学习 Netty 的时候看到这句话是没太能理解的。

说白了，`EventLoop` 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

那 `Channel` 和 `EventLoop` 直接有啥联系呢？

`Channel` 为 Netty 网络操作(读写等操作)抽象类，`EventLoop` 负责处理注册到其上的 `Channel` 处理 I/O 操作，两者配合参与 I/O 操作。

3.ChannelFuture

Netty 是异步非阻塞的，所有的 I/O 操作都为异步的。

因此，我们不能立刻得到操作是否执行成功，但是，你可以通过 `ChannelFuture` 接口的 `addListener()` 方法注册一个 `ChannelFutureListener`，当操作执行成功或者失败时，监听就会自动触发返回结果。

并且，你还可以通过 `ChannelFuture` 的 `channel()` 方法获取关联的 `Channel`

```
public interface ChannelFuture extends Future<Void> {
    Channel channel();

    ChannelFuture addListener(GenericFutureListener<? extends Future<? super
Void>> var1);
    .....

    ChannelFuture sync() throws InterruptedException;
}
```

另外，我们还可以通过 `ChannelFuture` 接口的 `sync()` 方法让异步的操作变成同步的。

4.ChannelHandler 和 ChannelPipeline

下面这段代码使用过 Netty 的小伙伴应该不会陌生，我们指定了序列化编解码器以及自定义的 `ChannelHandler` 处理消息。

```
b.group(eventLoopGroup)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) {
            ch.pipeline().addLast(new
NettyKryoDecoder(kryoSerializer, RpcResponse.class));
            ch.pipeline().addLast(new
NettyKryoEncoder(kryoSerializer, RpcRequest.class));
            ch.pipeline().addLast(new KryoClientHandler());
        }
    });
```

`ChannelHandler` 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

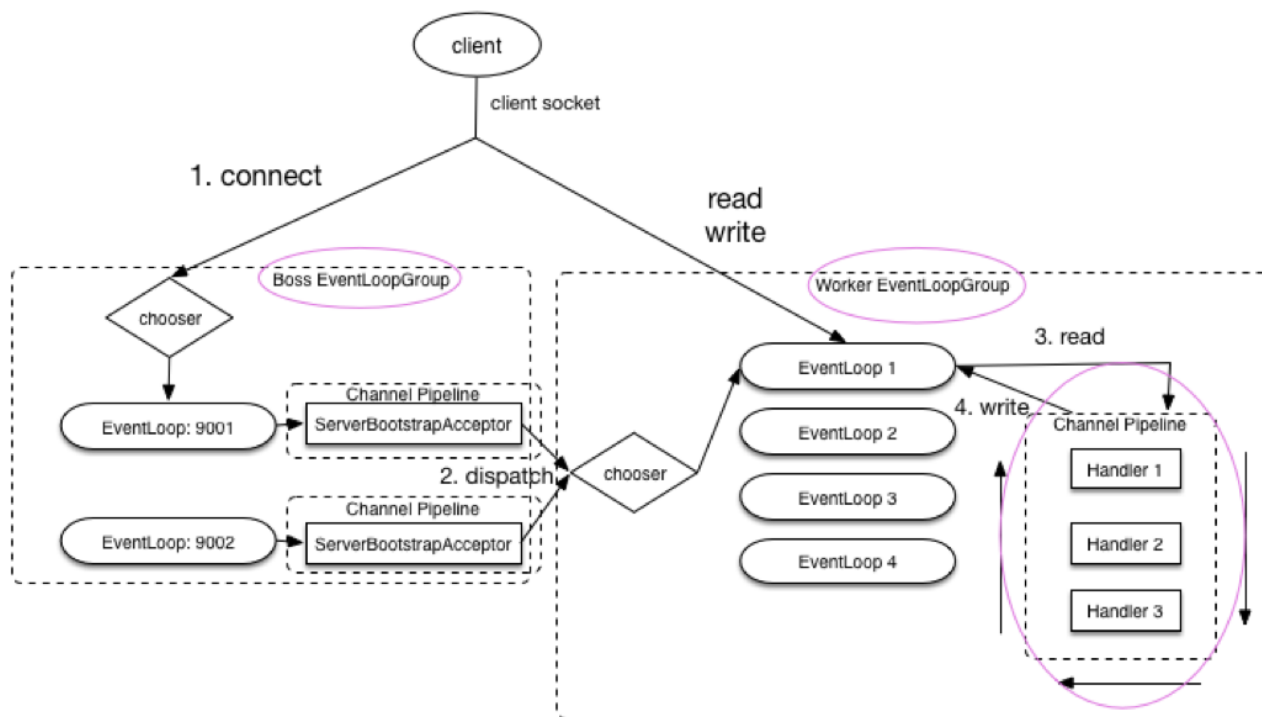
`ChannelPipeline` 为 `ChannelHandler` 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API。当 `Channel` 被创建时，它会被自动地分配到它专属的 `ChannelPipeline`。

我们可以在 `ChannelPipeline` 上通过 `addLast()` 方法添加一个或者多个 `ChannelHandler`，因为一个数据或者事件可能会被多个 Handler 处理。当一个 `ChannelHandler` 处理完之后就将数据交给下一个 `ChannelHandler`。

5.4.5 EventloopGroup 了解么?和 EventLoop 啥关系?

面试官：刚刚你也介绍了 EventLoop 。那你再说说 EventloopGroup 吧！和 EventLoop 啥关系？

我：



EventLoopGroup 包含多个 EventLoop（每一个 EventLoop 通常内部包含一个线程），上面我们已经说了 EventLoop 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

并且 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理，即 Thread 和 EventLoop 属于 1 : 1 的关系，从而保证线程安全。

上图是一个服务端对 EventLoopGroup 使用的大致模块图，其中 Boss EventloopGroup 用于接收连接，Worker EventloopGroup 用于具体的处理（消息的读写以及其他逻辑处理）。

从上图可以看出：当客户端通过 connect 方法连接服务端时，bossGroup 处理客户端连接请求。当客户端处理完成后，会将这个连接提交给 workerGroup 来处理，然后 workerGroup 负责处理其 IO 相关操作。

5.4.6 Bootstrap 和 ServerBootstrap 了解么？

 面试官：你再说说自己对 Bootstrap 和 ServerBootstrap 的了解吧！

 我：

Bootstrap 是客户端的启动引导类/辅助类，具体使用方法如下：

```
EventLoopGroup group = new NioEventLoopGroup();

try {
    //创建客户端启动引导/辅助类: Bootstrap
    Bootstrap b = new Bootstrap();
    //指定线程模型
    b.group(group).
        .....
    // 尝试建立连接
    ChannelFuture f = b.connect(host, port).sync();
    f.channel().closeFuture().sync();
} finally {
    // 优雅关闭相关线程组资源
    group.shutdownGracefully();
}
```

ServerBootstrap 客户端的启动引导类/辅助类，具体使用方法如下：

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();

try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup).
        .....
    // 6.绑定端口
    ChannelFuture f = b.bind(port).sync();
    // 等待连接关闭
    f.channel().closeFuture().sync();
} finally {
    //7.优雅关闭相关线程组资源
```

```
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}
```

从上面的示例中，我们可以看出：

1. `Bootstrap` 通常使用 `connect()` 方法连接到远程的主机和端口，作为一个 Netty TCP 协议通信中的客户端。另外，`Bootstrap` 也可以通过 `bind()` 方法绑定本地的一个端口，作为 UDP 协议通信中的一端。
2. `ServerBootstrap` 通常使用 `bind()` 方法绑定本地的端口上，然后等待客户端的连接。
3. `Bootstrap` 只需要配置一个线程组 — `EventLoopGroup`，而 `ServerBootstrap` 需要配置两个线程组 — `EventLoopGroup`，一个用于接收连接，一个用于具体的处理。

5.4.7 NioEventLoopGroup 默认的构造函数会起多少线程？

 面试官：看过 Netty 的源码了么？`NioEventLoopGroup` 默认的构造函数会起多少线程呢？

 我：嗯嗯！看过部分。

回顾我们在上面写的服务器端的代码：

```
// 1.bossGroup 用于接收连接，workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

为了搞清楚 `NioEventLoopGroup` 默认的构造函数到底创建了多少个线程，我们来看一下它的源码。

```
/**
 * 无参构造函数。
 * nThreads:0
 */
public NioEventLoopGroup() {
    //调用下一个构造方法
    this(0);
}
```

```

/**
 * Executor: null
 */
public NioEventLoopGroup(int nThreads) {
    //继续调用下一个构造方法
    this(nThreads, (Executor) null);
}

//中间省略部分构造函数

/**
 * RejectedExecutionHandler () : RejectedExecutionHandlers.reject()
 */
public NioEventLoopGroup(int nThreads, Executor executor, final
SelectorProvider selectorProvider,final SelectStrategyFactory
selectStrategyFactory) {
    //开始调用父类的构造函数
    super(nThreads, executor, selectorProvider, selectStrategyFactory,
RejectedExecutionHandlers.reject());
}

```

一直向下走下去的话，你会发现 `MultithreadEventLoopGroup` 类中有相关的指定线程数的代码，如下：

```

// 从1，系统属性，CPU核心数*2 这三个值中取出一个最大的
//可以得出 DEFAULT_EVENT_LOOP_THREADS 的值为CPU核心数*2
private static final int DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
NettyRuntime.availableProcessors() * 2));

// 被调用的父类构造函数，NioEventLoopGroup 默认的构造函数会起多少线程的秘密所在
// 当指定的线程数nThreads为0时，使用默认的线程数DEFAULT_EVENT_LOOP_THREADS
protected MultithreadEventLoopGroup(int nThreads, ThreadFactory
threadFactory, Object... args) {
    super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads,
threadFactory, args);
}

```

综上，我们发现 `NioEventLoopGroup` 默认的构造函数实际会起的线程数为 **CPU核心数*2**。

另外，如果你继续深入下去看构造函数的话，你会发现每个 `NioEventLoopGroup` 对象内部都会分配一组 `NioEventLoop`，其大小是 `nThreads`，这样就构成了一个线程池，一个 `NioEventLoop` 和一个线程相对应，这和我们上面说的 `EventloopGroup` 和 `EventLoop` 关系这部分内容相对应。

5.4.8 Netty 线程模型了解么？

 面试官：说一下 Netty 线程模型吧！

 我：大部分网络框架都是基于 Reactor 模式设计开发的。

Reactor 模式基于事件驱动，采用多路复用将事件分发给相应的 Handler 处理，非常适合处理海量 IO 的场景。

在 Netty 主要靠 `NioEventLoopGroup` 线程池来实现具体的线程模型的。

我们实现服务端的时候，一般会初始化两个线程组：

1. `bossGroup`：接收连接。
2. `workerGroup`：负责具体的处理，交由对应的 Handler 处理。

下面我们来详细看一下 Netty 中的线程模型吧！

1.单线程模型：

一个线程需要执行处理所有的 `accept`、`read`、`decode`、`process`、`encode`、`send` 事件。对于高负载、高并发，并且对性能要求比较高的场景不适用。

对应到 Netty 代码是下面这样的

使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 **CPU 核心数 *2**。

```
//1.eventGroup既用于处理客户端连接，又负责具体的处理。
EventLoopGroup eventGroup = new NioEventLoopGroup(1);
//2.创建服务端启动引导/辅助类：ServerBootstrap
ServerBootstrap b = new ServerBootstrap();
        bootstrap.group(eventGroup, eventGroup)
//.....
```

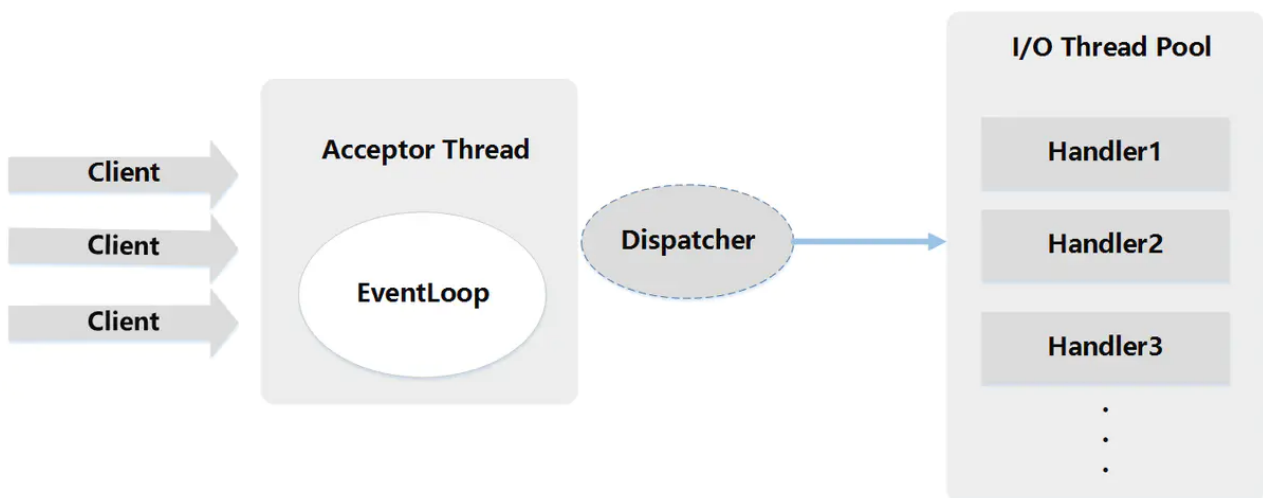
2.多线程模型

一个 Acceptor 线程只负责监听客户端的连接，一个 NIO 线程池负责具体处理：

accept 、 read 、 decode 、 process 、 encode 、 send 事件。满足绝大部分应用场景，并发连接量不大的时候没啥问题，但是遇到并发连接大的时候就可能会出现性能问题，成为性能瓶颈。

对应到 Netty 代码是下面这样的：

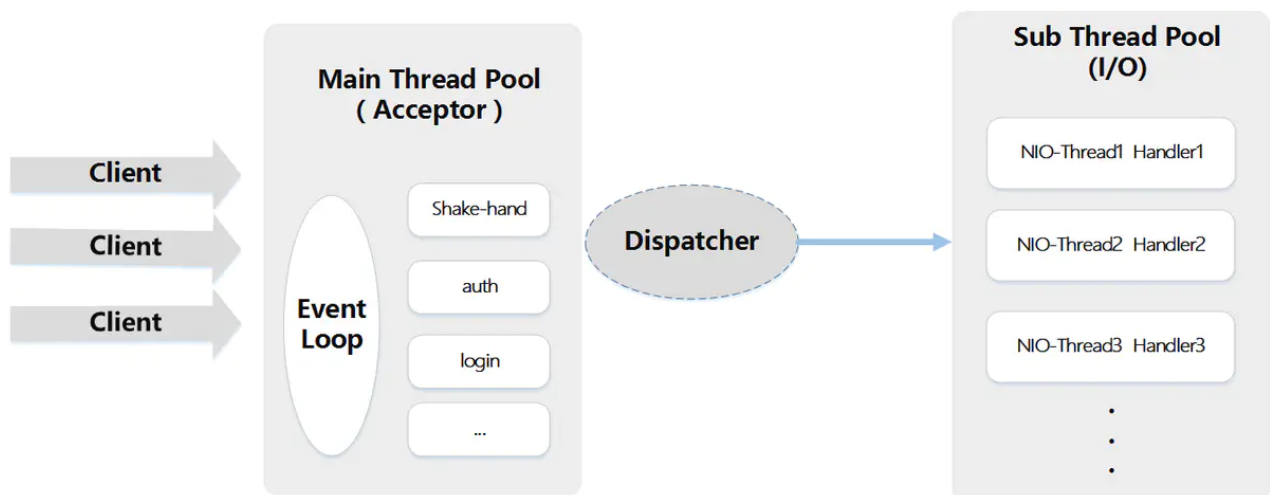
```
// 1.bossGroup 用于接收连接，workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类：ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组，确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....
```



3.主从多线程模型

从一个 主线程 NIO 线程池中选择一个线程作为 Acceptor 线程，绑定监听端口，接收客户端连接，其他线程负责后续的接入认证等工作。连接建立完成后，Sub NIO 线程池负责具体处理 I/O 读写。如果多线程模型无法满足你的需求的时候，可以考虑使用主从多线程模型。

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....
}
```



5.4.9 Netty 服务端和客户端的启动过程了解么？

服务端

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    // (非必备)打印日志
}
```

```

        .handler(new LoggingHandler(LogLevel.INFO))
        // 4.指定 IO 模型
        .channel(NioServerSocketChannel.class)
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) {
                ChannelPipeline p = ch.pipeline();
                //5.可以自定义客户端消息的业务处理逻辑
                p.addLast(new HelloServerHandler());
            }
        });
    // 6.绑定端口,调用 sync 方法阻塞知道绑定完成
    ChannelFuture f = b.bind(port).sync();
    // 7.阻塞等待直到服务器Channel关闭(closeFuture()方法获取Channel 的
    CloseFuture对象,然后调用sync()方法)
    f.channel().closeFuture().sync();
} finally {
    //8.优雅关闭相关线程组资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}

```

简单解析一下服务端的创建过程具体是怎样的：

1.首先你创建了两个 `NioEventLoopGroup` 对象实例：`bossGroup` 和 `workerGroup` 。

- `bossGroup` ：用于处理客户端的 TCP 连接请求。
- `workerGroup` ： 负责每一条连接的具体读写数据的处理逻辑，真正负责 I/O 读写操作，交由对应的 Handler 处理。

举个例子：我们把公司的老板当做 `bossGroup`，员工当做 `workerGroup`，`bossGroup` 在外面接完活之后，扔给 `workerGroup` 去处理。一般情况下我们会指定 `bossGroup` 的 线程数为 1（并发连接量不大的时候），`workGroup` 的线程数量为 **CPU 核心数 *2**。另外，根据源码来看，使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 **CPU 核心数 *2**。

2.接下来 我们创建了一个服务端启动引导/辅助类：`ServerBootstrap`，这个类将引导我们进行服务端的启动工作。

3.通过 `.group()` 方法给引导类 `ServerBootstrap` 配置两大线程组，确定了线程模型。

通过下面的代码，我们实际配置的是多线程模型，这个在上面提到过。

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

4.通过 `channel()` 方法给引导类 `ServerBootstrap` 指定了 IO 模型为 `NIO`

- `NioServerSocketChannel` : 指定服务端的 IO 模型为 `NIO`, 与 `BIO` 编程模型中的 `ServerSocket` 对应
- `NioSocketChannel` : 指定客户端的 IO 模型为 `NIO`, 与 `BIO` 编程模型中的 `Socket` 对应

5.通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer` , 然后指定了服务端消息的业务处理逻辑 `HelloServerHandler` 对象

6.调用 `ServerBootstrap` 类的 `bind()` 方法绑定端口

客户端

```
//1.创建一个 NioEventLoopGroup 对象实例
EventLoopGroup group = new NioEventLoopGroup();
try {
    //2.创建客户端启动引导/辅助类: Bootstrap
    Bootstrap b = new Bootstrap();
    //3.指定线程组
    b.group(group)
        //4.指定 IO 模型
        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws
Exception {
                ChannelPipeline p = ch.pipeline();
                // 5.这里可以自定义消息的业务处理逻辑
                p.addLast(new HelloClientHandler(message));
            }
        });
    // 6.尝试建立连接
    ChannelFuture f = b.connect(host, port).sync();
    // 7.等待连接关闭 (阻塞, 直到Channel关闭)
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

```
}
```

继续分析一下客户端的创建流程：

1. 创建一个 `NioEventLoopGroup` 对象实例
2. 创建客户端启动的引导类是 `Bootstrap`
3. 通过 `.group()` 方法给引导类 `Bootstrap` 配置一个线程组
4. 通过 `channel()` 方法给引导类 `Bootstrap` 指定了 IO 模型为 `NIO`
5. 通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer`，然后指定了客户端消息的业务处理逻辑 `HelloClientHandler` 对象
6. 调用 `Bootstrap` 类的 `connect()` 方法进行连接，这个方法需要指定两个参数：
 - `inetHost` : ip 地址
 - `inetPort` : 端口号

```
public ChannelFuture connect(String inetHost, int inetPort) {  
    return this.connect(InetSocketAddress.createUnresolved(inetHost,  
inetPort));  
}  
  
public ChannelFuture connect(SocketAddress remoteAddress) {  
    ObjectUtil.checkNotNull(remoteAddress, "remoteAddress");  
    this.validate();  
    return this.doResolveAndConnect(remoteAddress,  
this.config.localAddress());  
}
```

`connect` 方法返回的是一个 `Future` 类型的对象

```
public interface ChannelFuture extends Future<Void> {  
    .....  
}
```


也就是说这个方法是异步的，我们通过 `addListener` 方法可以监听到连接是否成功，进而打印出连

接信息。具体做法很简单，只需要对代码进行以下改动：

```
ChannelFuture f = b.connect(host, port).addListener(future -> {
    if (future.isSuccess()) {
        System.out.println("连接成功!");
    } else {
        System.err.println("连接失败!");
    }
}).sync();
```

5.4.10 什么是 TCP 粘包/拆包?有什么解决办法呢?

 面试官：什么是 TCP 粘包/拆包？

 **我：** TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。比如你多次发送：“你好,你真帅啊！哥哥！”，但是客户端接收到的可能是下面的：

[illegible]

 面试官：那有什么解决办法呢？

 我：

1.使用 Netty 自帶的解碼器

- **LineBasedFrameDecoder** : 发送端发送数据包的时候, 每个数据包之间以换行符作为分隔, **LineBasedFrameDecoder** 的工作原理是它依次遍历 **ByteBuf** 中的可读字节, 判断是否有换行符, 然后进行相应的截取。
- **DelimiterBasedFrameDecoder** : 可以自定义分隔符解码器, **LineBasedFrameDecoder** 实际上是一种特殊的 **DelimiterBasedFrameDecoder** 解码器。
- **FixedLengthFrameDecoder** : 固定长度解码器, 它能够按照指定的长度对消息进行相应的拆包。
- **LengthFieldBasedFrameDecoder** :

2.自定义序列化编解码器

在 Java 中自带的有实现 `Serializable` 接口来实现序列化，但由于它性能、安全性等原因一般情况下是不会被使用到的。


通常情况下，我们使用 Protostuff、Hessian2、json 序列方式比较多，另外还有一些序列化性能非常好的序列化方式也是很好的选择：

- 专门针对 Java 语言的：Kryo，FST 等等
- 跨语言的：Protostuff（基于 protobuf 发展而来），ProtoBuf，Thrift，Avro，MsgPack 等等

由于篇幅问题，这部分内容会在后续的文章中详细分析介绍~~~

5.4.11 Netty 长连接、心跳机制了解么？

 面试官：TCP 长连接和短连接了解么？

 我：我们知道 TCP 在进行读写之前，server 与 client 之间必须提前建立一个连接。建立连接的过程，需要我们常说的三次握手，释放/关闭连接的话需要四次挥手。这个过程是比较消耗网络资源并且有时间延迟的。

所谓，短连接说的就是 server 端与 client 端建立连接之后，读写完成之后就关闭掉连接，如果下一次再要互相发送消息，就要重新连接。短连接的有点很明显，就是管理和实现都比较简单，缺点也很明显，每一次的读写都要建立连接必然会带来大量网络资源的消耗，并且连接的建立也需要耗费时间。

长连接说的就是 client 向 server 双方建立连接之后，即使 client 与 server 完成一次读写，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。长连接的可以省去较多的 TCP 建立和关闭的操作，降低对网络资源的依赖，节约时间。对于频繁请求资源的客户来说，非常适用长连接。

 面试官：为什么需要心跳机制？Netty 中心跳机制了解么？

 我：

在 TCP 保持长连接的过程中，可能会出现断网等网络异常出现，异常发生的时候，client 与 server 之间如果没有交互的话，它们是无法发现对方已经掉线的。为了解决这个问题，我们就需要引入 **心跳机制**。

心跳机制的工作原理是：在 client 与 server 之间在一定时间内没有数据交互时，即处于 idle 状态时，客户端或服务器就会发送一个特殊的数据包给对方，当接收方收到这个数据报文后，也立即发送一个特殊的数据报文，回应发送方，此即一个 PING-PONG 交互。所以，当某一端收到心跳消息后，就知道了对方仍然在线，这就确保 TCP 连接的有效性。

TCP 实际上自带的就有长连接选项，本身是也有心跳包机制，也就是 TCP 的选项：`SO_KEEPALIVE`。但是，TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是应用层协议上实现自定义心跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 `IdleStateHandler`。

5.4.12 Netty 的零拷贝了解么？

 面试官：讲讲 Netty 的零拷贝？

 我：

维基百科是这样介绍零拷贝的：

零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。

在 OS 层面上的 `Zero-copy` 通常指避免在 `用户态(User-space)` 与 `内核态(Kernel-space)` 之间来回拷贝数据。而在 Netty 层面，零拷贝主要体现在对于数据操作的优化。

Netty 中的零拷贝体现在以下几个方面

1. 使用 Netty 提供的 `CompositeByteBuf` 类，可以将多个 `ByteBuf` 合并为一个逻辑上的 `ByteBuf`，避免了各个 `ByteBuf` 之间的拷贝。
2. `ByteBuf` 支持 `slice` 操作，因此可以将 `ByteBuf` 分解为多个共享同一个存储区域的 `ByteBuf`，避免了内存的拷贝。
3. 通过 `FileRegion` 包装的 `FileChannel.transferTo` 实现文件传输，可以直接将文件缓冲区的数据发送到目标 `Channel`，避免了传统通过循环 `write` 方式导致的内存拷贝问题。

参考

- netty 学习系列二：NIO Reactor 模型 & Netty 线程模型：<https://www.jianshu.com/p/38b56531565d>
- 《Netty 实战》
- Netty 面试题整理(2):<https://metatronxl.github.io/2019/10/22/Netty-面试题整理-二/>
- Netty (3) — 源码 NioEventLoopGroup:<https://www.cnblogs.com/qdhxhz/p/10075568.html>
- 对于 Netty ByteBuf 的零拷贝(Zero Copy) 的理解：
<https://www.cnblogs.com/xys1228/p/6088805.html>-----

#

5.5 SpringBoot面试题总结

概览（看看自己能回答几题）：

1. 简单介绍一下 Spring?有啥缺点?
2. 为什么要有 SpringBoot?
3. 说出使用 Spring Boot 的主要优点
4. 什么是 Spring Boot Starters?
5. Spring Boot 支持哪些内嵌 Servlet 容器?
6. 如何在 Spring Boot 应用程序中使用 Jetty 而不是 Tomcat?
7. 介绍一下@SpringBootApplication 注解
8. Spring Boot 的自动配置是如何实现的?
9. 开发 RESTful Web 服务常用的注解有哪些?
10. Spring Boot 常用的两种配置文件
11. 什么是 YAML? YAML 配置的优势在哪里?
12. Spring Boot 常用的读取配置文件的方法有哪些?
13. Spring Boot 加载配置文件的优先级了解么?
14. 常用的 Bean 映射工具有哪些?
15. Spring Boot 如何监控系统实际运行状况?
16. Spring Boot 如何做请求参数校验?
17. 如何使用 Spring Boot 实现全局异常处理?
18. Spring Boot 中如何实现定时任务?

答案地址：<https://t.zsxq.com/Uv3ByZn>。这部分内容的答案更新在了[知识星球](#)。