Spring 框架中都用到了哪些设计模式?

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：

- 代理模式 — 在 AOP 和 remoting 中被用的比较多。
- 单例模式 — 在 Spring 配置文件中定义的 Bean 默认为单例模式。
- 模板方法 — 用来解决代码重复的问题。比如 RestTemplate、JmsTemplate、JdbcTemplate 。
- 前端控制器 — Spring提供了 DispatcherServlet 来对请求进行分发。
- 视图帮助(View Helper) — Spring 提供了一系列的 JSP 标签，高效宏来辅助将分散的代码整合在视图里。
- 依赖注入 — 贯穿于 BeanFactory / ApplicationContext 接口的核心理念。
- 工厂模式 — BeanFactory 用来创建对象的实例。

当然，感兴趣的胖友，觉得不过瘾，可以看看芳芳基友知秋写的几篇文章：

- 《Spring 框架中的设计模式(一)》
- 《Spring 框架中的设计模式(二)》
- 《Spring 框架中的设计模式(三)》
- 《Spring 框架中的设计模式(四)》
- 《Spring 框架中的设计模式(五)》

Spring IoC

下面，我们会将分成 IoC 和 Bean 两部分来分享 Spring 容器的内容。

- IoC ，侧重在于容器。
- Bean ，侧重在于被容器管理的 Bean 。

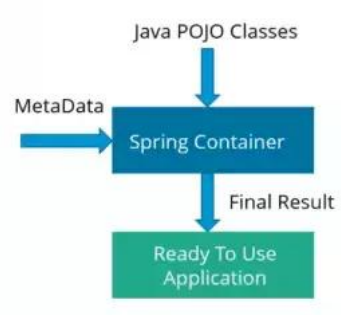
什么是 Spring IoC 容器?

注意，正确的拼写是 IoC 。

Spring 框架的核心是 Spring IoC 容器。容器创建 Bean 对象，将它们装配在一起，配置它们并管理它们的完整生命周期。

- Spring 容器使用**依赖注入**来管理组成应用程序的 Bean 对象。
- 容器通过读取提供的**配置元数据** Bean Definition 来接收对象进行实例化，配置和组装的指令。
- 该配置元数据 Bean Definition 可以通过 XML，Java 注解或 Java Config 代码**提供**。

芳芳，注意上面三段段话的**加粗**部分的内容。



什么是依赖注入?

在依赖注入中，你不必主动、手动创建对象，但必须描述如何创建它们。

- 你不是直接在代码中将组件和服务连接在一起，而是描述配置文件中哪些组件需要哪些服务。
- 然后，再由 IoC 容器将它们装配在一起。

另外，依赖注入的英文缩写是 Dependency Injection ，简称 DI 。

IoC 和 DI 有什么区别?

芳芳的吐槽，最怕这种概念题。下面引用知乎上的一个讨论：《IoC 和DI 有什么区别？》

IoC 是个更宽泛的概念，DI 是更具体的。引用郑烨的一篇博客，引用郑烨的一篇博客，我眼中的Spring

> **Dependency Injection**
>
> 原来，它叫 IoC 。
>
> Martin Flower 发话了，是个框架都有 IoC ，这不足以新生容器反转的“如何定位插件的具体实现”，于是，它有了个新名字，Dependency Injection 。
>
> 其实，它就是一种将调用者与被调用者分离的思想，Uncle Bob 管它叫DIP（Dependency Inversion Principle），并把它归入OO设计原则。
>
> 同 Spring 相比，它更早进入我的大脑。一切都是那么朦胧，直至 Spring 出现。

慢慢的，我知道了它还分为三种：

- Interface Injection（type 1）
- Setter Injection（type 2）
- Constructor Injection（type 3）。

Martin Flower那篇为它更名的大作让我心目关于它的一切趋于完整。

在 Spring 中，它是一切的基础。Spring 的种种优势随之而来。

于我而言，它为我带来更多的是思维方式的转变，恐怕以后我再也无法写出那种一大块的全功能程序了。

可以通过多少种方式完成依赖注入？

通常，依赖注入可以通过**三种**方式完成，即：

上面一个问题的三种方式的英文，下面是三种方式的中文。

- 接口注入
- 构造函数注入
- setter 注入

目前，在 Spring Framework 中，仅使用构造函数和 setter 注入这**两种**方式。

那么这两种方式各有什么优缺点呢？胖友可以简单阅读 《Spring两种依赖注入方式的比较》，不用太较真。综述来说：

构造函数注入	setter 注入
没有部分注入	有部分注入
不会覆盖 setter 属性	会覆盖 setter 属性
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

- 实际场景下，setting 注入使用的更多。

Spring 中有多少种 IoC 容器？

Spring 提供了两种(不是“个”) IoC 容器，分别是 BeanFactory、ApplicationContext 。

BeanFactory

BeanFactory 在 `spring-beans` 项目提供。

BeanFactory ，就像一个包含 Bean 集合的工厂类。它会在客户端要求时实例化 Bean 对象。

ApplicationContext

ApplicationContext 在 `spring-context` 项目提供。

ApplicationContext 接口扩展了 BeanFactory 接口，它在 BeanFactory 基础上提供了一些额外的功能。内置如下功能：

- MessageSource： 管理 message ，实现国际化等功能。
- ApplicationEventPublisher： 事件发布。
- ResourcePatternResolver： 多资源加载。
- EnvironmentCapable： 系统 Environment（profile + Properties）相关。
- Lifecycle： 管理生命周期。
- Closable： 关闭，释放资源
- InitializingBean：自定义初始化。
- BeanNameAware：设置 beanName 的 Aware 接口。

另外，ApplicationContext 会自动初始化非懒加载的 Bean 对象们。

详细的内容，感兴趣的胖友，可以看看 《【死磕 Spring】—— ApplicationContext 相关接口架构分析》 一文。源码之前无秘密。简单总结下 BeanFactory 与 ApplicationContext 两者的差异：

芳芳：可能很多胖友没看过源码，所以会比较难。

BeanFactory	ApplicationContext
它使用懒加载	它使用即时加载
它使用语法显式提供资源对象	它自己创建和管理资源对象
不支持国际化	支持国际化
不支持基于依赖的注解	支持基于依赖的注解

另外，BeanFactory 也被称为**低级**容器，而 ApplicationContext 被称为**高级**容器。

请介绍下常用的 BeanFactory 容器?

BeanFactory 最常用的是 XmlBeanFactory 。它可以根据 XML 文件中定义的内容，创建相应的 Bean。

请介绍下常用的 ApplicationContext 容器?

以下是三种较常见的 ApplicationContext 实现方式：

- 1、ClassPathXmlApplicationContext： 从 ClassPath 的 XML 配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。示例代码如下：

```
ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```

- 2、FileSystemXmlApplicationContext： 由文件系统中的XML配置文件读取上下文。示例代码如下：

```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```

- 3、XmlWebApplicationContext： 由 Web 应用的XML文件读取上下文。例如我们在 Spring MVC 使用的情况。

当然，目前我们更多的是使用 Spring Boot 为主，所以使用的是第四种 ApplicationContext 容器，ConfigServletWebServerApplicationContext 。

列举一些 IoC 的一些好处?

- 它将最小化应用程序中的代码量。
- 它以最小的影响和最少的侵入机制促进松耦合。
- 它支持即时的实例化和延迟加载 Bean 对象。
- 它将使您的应用程序易于测试，因为它不需要单元测试用例中的任何单例或 JNDI 查找机制。

简述 Spring IoC 的实现机制?

简单来说，Spring 中的 IoC 的实现原理，就是**工厂模式**加**反射机制**。代码如下：

```
interface Fruit {

public abstract void eat();
}
class Apple implements Fruit {

public void eat(){
System.out.println("Apple");
    }
}
class Orange implements Fruit {

public void eat(){
System.out.println("Orange");
    }
}
```

```
class Factory {

    public static Fruit getInstance(String className) {
        Fruit f = null;
        try {
            f = (Fruit) Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return f;
    }
}

class Client {

    public static void main(String[] args) {
        Fruit f = Factory.getInstance("io.github.dunwu.spring.Apple");

        if(f != null){
            f.eat();
        }
    }
}
```

- Fruit 接口，有 Apple 和 Orange 两个实现类。
- Factory 工厂，通过反射机制，创建 `className` 对应的 Fruit 对象。
- Client 通过 Factory 工厂，获得对应的 Fruit 对象。
- 😈 实际情况下，Spring IoC 比这个复杂很多很多，例如单例 Bean 对象，Bean 的属性注入，相互依赖的 Bean 的处理，以及等等。

在基友 《面试问烂的 Spring IoC 过程》 的文章中，把 Spring IoC 相关的内容，讲的非常不错。

Spring 框架中有哪些不同类型的事件?

Spring 的 ApplicationContext 提供了支持事件和代码中监听器的功能。

我们可以创建 Bean 用来监听在 ApplicationContext 中发布的事件。如果一个 Bean 实现了 ApplicationListener 接口，当一个ApplicationEvent 被发布以后，Bean 会自动被通知。示例代码如下：

```
public class AllApplicationEventListener implements ApplicationListener<ApplicationEvent> {

@Override
public void onApplicationEvent(ApplicationEvent applicationEvent) {
//process event
    }
}
```

Spring 提供了以下五种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：该事件会在ApplicationContext 被初始化或者更新时发布。也可以在调用ConfigurableApplicationContext 接口中的 `#refresh()` 方法时被触发。
2. 上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext 的 `#start()` 方法开始/重新开始容器时触发该事件。
3. 上下文停止事件（ContextStoppedEvent）：当容器调用 ConfigurableApplicationContext 的 `#stop()` 方法停止容器时触发该事件。
4. 上下文关闭事件（ContextClosedEvent）：当ApplicationContext 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
5. 请求处理事件（RequestHandledEvent）：在 We b应用中，当一个HTTP 请求（request）结束触发该事件。

除了上面介绍的事件以外，还可以通过扩展 ApplicationEvent 类来开发**自定义**的事件。

① 示例自定义的事件的类，代码如下：

```
public class CustomApplicationEvent extends ApplicationEvent{

public CustomApplicationEvent(Object source, final String msg) {
super(source);
    }

}
```

② 为了监听这个事件，还需要创建一个监听器。示例代码如下：

```
public class CustomEventListener implements ApplicationListener<CustomApplicationEvent> {

    @Override
    public void onApplicationEvent(CustomApplicationEvent applicationEvent) {
        // handle event
    }
}
```

③ 之后通过 ApplicationContext 接口的 `#publishEvent(Object event)` 方法，来发布自定义事件。示例代码如下：

```
// 创建 CustomApplicationEvent 事件
CustomApplicationEvent customEvent = new CustomApplicationEvent(applicationContext, "Test message");
// 发布事件
applicationContext.publishEvent(customEvent);
```

Spring Bean

什么是 Spring Bean？

- Bean 由 Spring IoC 容器实例化，配置，装配和管理。
- Bean 是基于用户提供给 IoC 容器的配置元数据 Bean Definition 创建。

这个问题，胖友可以在回过头看 「什么是 Spring IoC 容器？」 问题，相互对照。

Spring 有哪些配置方式

单纯从 Spring Framework 提供的方式，一共有三种：

- 1、XML 配置文件。

Bean 所需的依赖项和服务在 XML 格式的配置文件中指定。这些配置文件通常包含许多 bean 定义和特定于应用程序的配置选项。它们通常以 bean 标签开头。例如：

```
<bean
id="stude
ntBean"
class="or
g.edureka
.firstSpr
ing.Stude
ntBean">
<property
name="nam
e"
value="Ed
ureka">
</propert
y>
</bean>
```

- 2、注解配置。

您可以通过在相关的类，方法或字段声明上使用注解，将 Bean 配置为组件类本身，而不是使用 XML 来描述 Bean 装配。默认情况下，Spring 容器中未打开注解装配。因此，您需要在使用它之前在 Spring 配置文件中启用它。例如：

```
<beans>
<context:
annotatio
n-
config/>
<!-- bean
definitio
ns go
here -->
</beans>
```

- 3、Java Config 配置。

Spring 的 Java 配置是通过使用 @Bean 和 @Configuration 来实现。

  - `@Bean` 注解扮演与 `<bean />` 元素相同的角色。
  - `@Configuration` 类允许通过简单地调用同一个类中的其他 `@Bean` 方法来定义 Bean 间依赖关系。
  - 例如：

```
@Configur
ation
public
class
StudentCo
nfig {

    @Bean

public
StudentBe
an
myStudent
() {

return
new
StudentBe
an();
    }

}
```

    - 是不是很熟悉 😈

目前主要使用 **Java Config** 配置为主。当然，三种配置方式是可以混合使用的。例如说：

- Dubbo 服务的配置，艿艿喜欢使用 XML 。
- Spring MVC 请求的配置，艿艿喜欢使用 `@RequestMapping` 注解。
- Spring MVC 拦截器的配置，艿艿喜欢 Java Config 配置。

另外，现在已经是 Spring Boot 的天下，所以更加是 **Java Config** 配置为主。

Spring 支持几种 Bean Scope？

艿艿，这个是一个比较小众的题目，简单了解即可。

Spring Bean 支持 5 种 Scope ，分别如下：

- Singleton - 每个 Spring IoC 容器仅有一个单 Bean 实例。**默认**
- Prototype - 每次请求都会产生一个新的实例。
- Request - 每一次 HTTP 请求都会产生一个新的 Bean 实例，并且该 Bean 仅在当前 HTTP 请求内有效。
- Session - 每一个的 Session 都会产生一个新的 Bean 实例，同时该 Bean 仅在当前 HTTP Session 内有效。
- Application - 每一个 Web Application 都会产生一个新的 Bean ，同时该 Bean 仅在当前 Web Application 内有效。

另外，网络上很多文章说有 Global-session 级别，它是 Portlet 模块独有，目前已经废弃，在 Spring5 中是找不到的。

仅当用户使用支持 Web 的 ApplicationContext 时，**最后三个才可用**。

再补充一点，开发者是可以**自定义** Bean Scope ，具体可参见 《Spring（10）—— Bean 作用范围（二）—— 自定义 Scope》 。

不错呢，还是那句话，这个题目简单了解下即可，实际常用的只有 Singleton 和 Prototype 两种级别，甚至说，只有 Singleton 级别。😈

Spring Bean 在容器的生命周期是什么样的?

芳芳说：这是一个比较高级的 Spring 的面试题，非常常见，并且答对比较加分。当然，如果实际真正弄懂，需要对 Spring Bean 的源码，有比较好的理解，所以 《精尽 Spring 源码》 系列，该读还是读吧。

芳芳：要注意下面每段话，芳芳进行加粗的地方。

Spring Bean 的**初始化**流程如下：

- 实例化 Bean 对象
  - Spring 容器根据配置中的 Bean Definition(定义)中**实例化** Bean 对象。

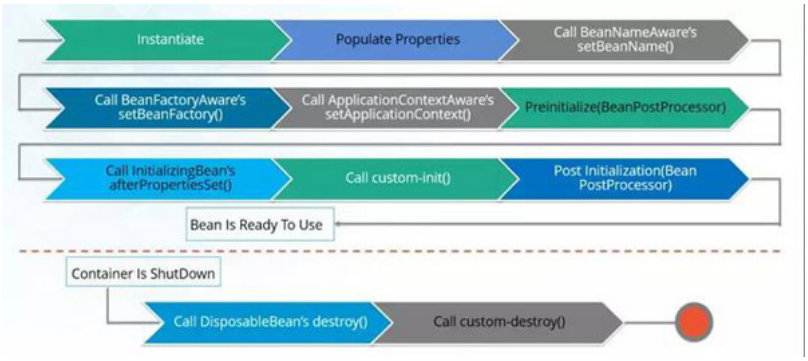
Bean Definition 可以通过 XML，Java 注解或 Java Config 代码提供。

  - Spring 使用依赖注入**填充**所有属性，如 Bean 中所定义的配置。
- Aware 相关的属性，注入到 Bean 对象
  - 如果 Bean 实现 **BeanNameAware** 接口，则工厂通过传递 Bean 的 beanName 来调用 `#setBeanName(String name)` 方法。
  - 如果 Bean 实现 **BeanFactoryAware** 接口，工厂通过传递自身的实例来调用 `#setBeanFactory(BeanFactory beanFactory)` 方法。
- 调用相应的方法，进一步初始化 Bean 对象
  - 如果存在与 Bean 关联的任何 **BeanPostProcessor** 们，则调用 `#preProcessBeforeInitialization(Object bean, String beanName)` 方法。
  - 如果 Bean 实现 **InitializingBean** 接口，则会调用 `#afterPropertiesSet()` 方法。
  - 如果为 Bean 指定了 **init** 方法（例如 `<bean />` 的 `init-method` 属性），那么将调用该方法。
  - 如果存在与 Bean 关联的任何 **BeanPostProcessor** 们，则将调用 `#postProcessAfterInitialization(Object bean, String beanName)` 方法。

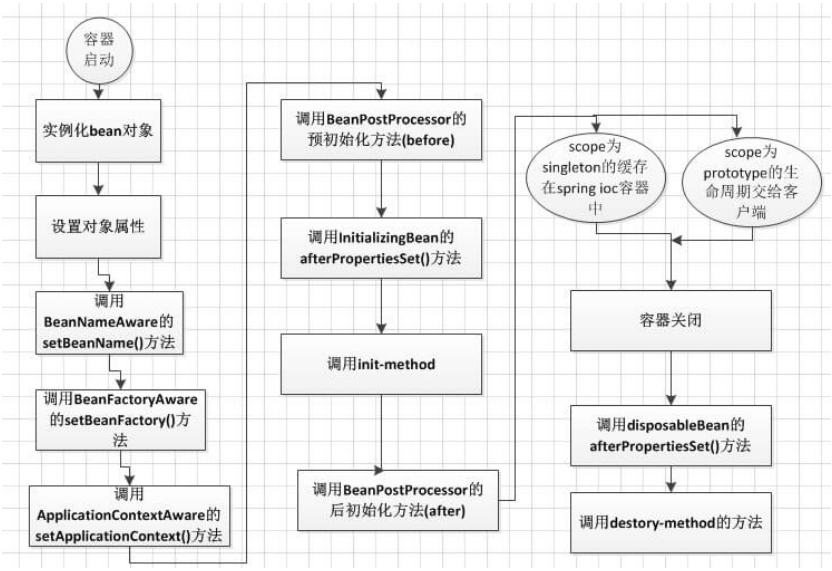
Spring Bean 的**销毁**流程如下：

- 如果 Bean 实现 **DisposableBean** 接口，当 spring 容器关闭时，会调用 `#destroy()` 方法。
- 如果为 bean 指定了 **destroy** 方法（例如 `<bean />` 的 `destroy-method` 属性），那么将调用该方法。

整体如下图：



无意中，芳芳又翻到一张有趣的整体图，如下图：



什么是 Spring 的内部 bean?

只有将 Bean **仅**用作另一个 Bean 的属性时，才能将 Bean 声明为内部 Bean。

- 为了定义 Bean，Spring 提供基于 XML 的配置元数据在 `<property>`或 `<constructor-arg>` 中提供了 `<bean>`元素的使用。
- 内部 Bean 总是**匿名**的，并且它们总是作为**原型 Prototype** 。

例如，假设我们有一个 Student 类，其中引用了 Person 类。这里我们将只创建一个 Person 类实例并在 Student 中使用它。示例代码如下：

```
// Student.java

public class Student {

    private Person person;
    // ... Setters and Getters
}

// Person.java

public class Person {

    private String name;
    private String address;
    // ... Setters and Getters
}
```

```
<!--
bean.xml
-->

<bean
id="Stude
ntBean"
class="co
m.edureka
.Student"
>

<property
name="per
son">

<!--This
is inner
bean -->

<bean
class="co
m.edureka
.Person">

<property
name="nam
e"
value="Sc
ott">
</propert
y>

<property
name="add
ress"
value="Ba
ngalore">
</propert
y>

</bean>

</propert
y>
</bean>
```

什么是 Spring 装配?

当 Bean 在 Spring 容器中组合在一起时，它被称为**装配**或 **Bean 装配**。Spring 容器需要知道需要什么 Bean 以及容器应该如何使用依赖注入来将 Bean 绑定在一起，同时装配 Bean 。

装配，和上文提到的 DI 依赖注入，实际是一个东西。

自动装配有哪些方式?

Spring 容器能够自动装配 Bean 。也就是说，可以通过检查 BeanFactory 的内容让 Spring 自动解析 Bean 的协作者。

自动装配的不同模式：

- no - 这是默认设置，表示没有自动装配。应使用显式 Bean 引用进行装配。
- byName - 它根据 Bean 的名称注入对象依赖项。它匹配并装配其属性与 XML 文件中由相同名称定义的 Bean 。
- 【最常用】**byType** - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 Bean 类型匹配，则匹配并装配属性。
- 构造函数 - 它通过调用类的构造函数来注入依赖项。它有大量的参数。
- autodetect - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

自动装配有什么局限?

芳芳：这个题目，了解下即可，也不是很准确。

- 覆盖的可能性 - 您始终可以使用 `<constructor-arg>` 和 `<property>` 设置指定依赖项，这将覆盖自动装配。
- 基本元数据类型 - 简单属性（如原数据类型，字符串和类）无法自动装配。

这种，严格来说，也不能称为局限。因为可以通过配置文件来解决。

- 令人困惑的性质 - 总是喜欢使用明确的装配，因为自动装配不太精确。

解释什么叫延迟加载?

默认情况下，容器启动之后会将所有作用域为**单例**的 Bean 都创建好，但是有的业务场景我们并不需要它提前都创建好。此时，我们可以在Bean 中设置 `lzay-init = "true"` 。

- 这样，当容器启动之后，作用域为单例的 Bean ，就不在创建。
- 而是在获得该 Bean 时，才真正在创建加载。

Spring 框架中的单例 Bean 是线程安全的么?

Spring 框架并没有对单例 Bean 进行任何多线程的封装处理。

- 关于单例 Bean 的线程安全和并发问题，需要开发者自行去搞定。

- 并且，单例的线程安全问题，也不是 Spring 应该去关心的。Spring 应该做的是，提供根据配置，创建单例 Bean 或多例 Bean 的功能。

当然，但实际上，大部分的 Spring Bean 并没有可变的状态(比如Serview 类和 DAO 类)，所以在某种程度上说 Spring 的单例 Bean 是线程安全的。

如果你的 Bean 有多种状态的话，就需要自行保证线程安全。最浅显的解决办法，就是将多态 Bean 的作用域(Scope)由 Singleton 变更为 Prototype 。

Spring Bean 怎么解决循环依赖的问题?

芳芳说：能回答出这个问题的，一般是比较厉害的。

这是个比较复杂的问题，有能力的胖友，建议看下 《【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（五）之循环依赖处理》

感觉，不通过源码，很难解释清楚这个问题。如果看不懂的胖友，可以在认真看完，在星球里，我们一起多交流下。好玩的。

Spring 注解

这块内容，实际写在 「Spring Bean」 中比较合适，考虑到后续的问题，都是关于注解的，所以单独起一个大的章节。

什么是基于注解的容器配置?

不使用 XML 来描述 Bean 装配，开发人员通过在相关的类，方法或字段声明上使用**注解**将配置移动到组件类本身。它可以作为 XML 设置的替代方案。例如：

Spring 的 Java 配置是通过使用 `@Bean` 和 `@Configuration` 来实现。

- `@Bean` 注解，扮演与 `<bean />` 元素相同的角色。
- `@Configuration` 注解的类，允许通过简单地调用同一个类中的其他 `@Bean` 方法来定义 Bean 间依赖关系。

示例如下：

```
@Configuration
public class StudentConfig {

    @Bean
public StudentBean myStudent() {
return new StudentBean();
    }
}
```

如何在 Spring 中启动注解装配?

默认情况下，Spring 容器中未打开注解装配。因此，要使用基于注解装配，我们必须通过配置 `<context: annotation-config />` 元素在 Spring 配置文件中启用它。

当然，如果胖友是使用 Spring Boot ，默认情况下已经开启。

@Component, @Controller, @Repository, @Service 有何区别?

- `@Component`： 它将 Java 类标记为 Bean 。它是任何 Spring 管理组件的**通用**构造型。
- `@Controller`： 它将一个类标记为 Spring Web MVC **控制器**。
- `@Service`： 此注解是组件注解的特化。它不会对 `@Component` 注解提供任何其他行为。您可以在**服务层**类中使用 @Service 而不是 `@Component`，因为它以更好的方式指定了意图。
- `@Repository`： 这个注解是具有类似用途和功能的 `@Component` 注解的特化。它为 **DAO** 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException 。

@Required 注解有什么用?

`@Required` 注解，应用于 Bean 属性 setter 方法。

- 此注解仅指示必须在配置时使用 Bean 定义中的显式属性值或使用自动装配填充受影响的 Bean 属性。
- 如果尚未填充受影响的 Bean 属性，则容器将抛出 BeanInitializationException 异常。

示例代码如下：

```
public class Employee {

    private String name;

    @Required
    public void setName(String name){
        this.name=name;
    }

    public string getName(){
        return name;
    }
}
```

- T T 貌似平时很少用这个注解噢。

@Autowired 注解有什么用?

`@Autowired` 注解，可以更准确地控制应该在何处以及如何进行自动装配。

- 此注解用于在 setter 方法，构造函数，具有任意名称或多个参数的属性或方法上自动装配 Bean。
- 默认情况下，它是类型驱动的注入。

示例代码如下：

```
public class EmpAccount {

    @Autowired
    private Employee emp;
}
```

@Qualifier 注解有什么用?

当你创建多个**相同类型**的 Bean ， 并希望仅使用属性装配**其中一个** Bean 时，您可以使用 `@Qualifier` 注解和 `@Autowired` 通过指定 ID 应该装配哪个**确切的** Bean 来消除歧义。

例如，应用中有两个类型为 Employee 的 Bean ID 为 `"emp1"` 和 `"emp2"`，此处，我们希望 EmployeeAccount Bean 注入 `"emp1"` 对应的 Bean 对象。代码如下：

```
public class EmployeeAccount {

    @Autowired
    @Qualifier(emp1)
    private Employee emp;

}
```

Spring AOP

Spring AOP 的面试题中，大多数都是概念题，主要是对切面的理解。概念点主要有：

- AOP

- `Aspect`
- `JoinPoint`
- `PointCut`
- `Advice`
- `Target`
- `AOP Proxy`
- `Weaving`

- 在阅读完 「Spring AOP」 的面试题后，在回过头思考下这些概念点，到底理解了多少。注意，不是背，理解！

非常推荐阅读如下两篇文章：

- 《彻底征服 Spring AOP 之理论篇》
- 《彻底征服 Spring AOP 之实战篇》

什么是 AOP？

AOP(Aspect-Oriented Programming)，即**面向切面编程**, 它与 OOP(Object-Oriented Programming, 面向对象编程) 相辅相成， 提供了与 OOP 不同的抽象软件结构的视角。

- 在 OOP 中，以类(Class)作为基本单元
- 在 AOP 中，以**切面(Aspect)**作为基本单元。

什么是 Aspect？

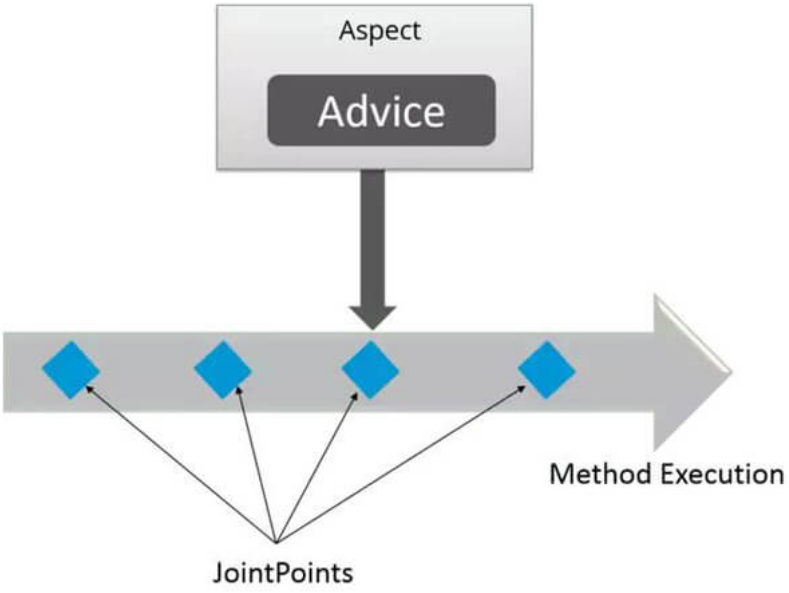
Aspect 由 **PointCut** 和 **Advice** 组成。

- 它既包含了横切逻辑的定义，也包括了连接点的定义。
- Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑编织到切面所指定的连接点中。

AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:

1. 如何通过 PointCut 和 Advice 定位到特定的 **JoinPoint** 上。
2. 如何在 Advice 中编写切面代码。

可以简单地认为, 使用 @Aspect 注解的类就是切面



什么是 JoinPoint？

JoinPoint ， **切点**，程序运行中的一些时间点, 例如：

- 一个方法的执行。
- 或者是一个异常的处理。

在 Spring AOP 中，JoinPoint 总是方法的执行点。

什么是 PointCut？

PointCut ， **匹配** JoinPoint 的谓词(a predicate that matches join points)。

简单来说，PointCut 是匹配 JoinPoint 的条件。

- Advice 是和特定的 PointCut 关联的，并且在 PointCut 相匹配的 JoinPoint 中执行。即 `Advice => PointCut => JoinPoint` 。
- 在 Spring 中, 所有的方法都可以认为是 JoinPoint ，但是我们并不希望在所有的方法上都添加 Advice 。**而 PointCut 的作用**，就是提

供一组规则(使用 AspectJ PointCut expression language 来描述) 来匹配 JoinPoint ，给满足规则的 JoinPoint 添加 Advice 。

😈 是不是觉得有点绕，实际场景下，其实也不会弄的这么清楚~~

关于 JoinPoint 和 PointCut 的区别

JoinPoint 和 PointCut 本质上就是**两个不同纬度上**的东西。

- 在 Spring AOP 中，所有的方法执行都是 JoinPoint 。而 PointCut 是一个描述信息，它修饰的是 JoinPoint ，通过 PointCut ，我们就可以确定哪些 JoinPoint 可以被织入 Advice 。
- Advice 是在 JoinPoint 上执行的，而 PointCut 规定了哪些 JoinPoint 可以执行哪些 Advice 。

或者，我们在换一种说法：

1. 首先，Advice 通过 PointCut 查询需要被织入的 JoinPoint 。
2. 然后，Advice 在查询到 JoinPoint 上执行逻辑。

什么是 Advice ?

Advice ， **通知**。

- 特定 JoinPoint 处的 Aspect 所采取的动作称为 Advice 。
- Spring AOP 使用一个 Advice 作为拦截器，在 JoinPoint “周围”维护一系列的**拦截器**。

有哪些类型的 Advice?

- Before - 这些类型的 Advice 在 JoinPoint 方法之前执行，并使用 `@Before` 注解标记进行配置。
- After Returning - 这些类型的 Advice 在连接点方法正常执行后执行，并使用 `@AfterReturning` 注解标记进行配置。
- After Throwing - 这些类型的 Advice 仅在 JoinPoint 方法通过抛出异常退出并使用 `@AfterThrowing` 注解标记配置时执行。
- After Finally - 这些类型的 Advice 在连接点方法之后执行，无论方法退出是正常还是异常返回，并使用 `@After` 注解标记进行配置。
- Around - 这些类型的 Advice 在连接点之前和之后执行，并使用 `@Around` 注解标记进行配置。

😈 看起来，是不是和拦截器的执行时间，有几分相似。实际上，用于拦截效果的各种实现，大体都是类似的。

什么是 Target ?

Target ， 织入 Advice 的**目标对象**。目标对象也被称为 **Advised Object** 。

- 因为 Spring AOP 使用运行时代理的方式来实现 Aspect ，因此 Advised Object 总是一个代理对象(Proxied Object) 。
- **注意, Advised Object 指的不是原来的对象，而是织入 Advice 后所产生的代理对象**。
- Advice + Target Object = Advised Object = Proxy 。

AOP 有哪些实现方式?

实现 AOP 的技术，主要分为两大类：

- ① **静态代理** - 指使用 AOP 框架提供的命令进行编译，从而在编译阶段就可生成 AOP 代理类，因此也称为编译时增强。
  - 编译时编织（特殊编译器实现）
  - 类加载时编织（特殊的类加载器实现）。

  例如，SkyWalking 基于 Java Agent 机制，配置上 ByteBuddy 库，实现类加载时编织时增强，从而实现链路追踪的透明埋点。感兴趣的胖友，可以看看 《SkyWalking 源码分析之 JavaAgent 工具 ByteBuddy 的应用》 。
- ② **动态代理** - 在运行时在内存中“临时”生成 AOP 动态代理类，因此也被称为运行时增强。目前 Spring 中使用了两种动态代理库：
  - JDK 动态代理
  - CGLIB

那么 Spring 什么时候使用 JDK 动态代理，什么时候使用 CGLIB 呢?

```
// From 《Spring 源码深度解析》P172
// Spring AOP 部分使用 JDK 动态代理或者 CGLIB 来为目标对象创建代理。(建议尽量使用 JDK 的动态代理)
// 如果被代理的目标对象实现了至少一个接口，则会使用 JDK 动态代理。所有该目标类型实现的接口都将被代
```

```
理。
// 若该目标对象没有实现任何接口，则创建一个 CGLIB 代理。
// 如果你希望强制使用 CGLIB 代理，（例如希望代理目标对象的所有方法，而不只是实现自接口的方法）那也可以。但是需要考虑以下两个方法：
//
1> 无法通知 (advise) Final 方法，因为它们不能被覆盖。
//
2> 你需要将 CGLIB 二进制发型包放在 classpath 下面。
// 为什么 Spring 默认使用 JDK 的动态代理呢？笔者猜测原因如下：
//
1> 使用 JDK 原生支持，减少三方依赖
//
2> JDK8 开始后，JDK 代理的性能差距 CGLIB 的性能不会太多。可参见：https://www.cnblogs.com/haig/p/4304615.html
```

- 实际上，Spring AOP 的代码量不大，与其在窗户外面不清不楚，不如捅破它！感兴趣的胖友，可以撸一撸 《精尽 Spring 源码分析 —— AOP 源码简单导读》 。

或者，我们来换一个解答答案：

Spring AOP 中的动态代理主要有两种方式，

- JDK 动态代理

JDK 动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是 InvocationHandler 接口和 Proxy 类。

- CGLIB 动态代理

如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。当然，Spring 也支持配置，**强制**使用 CGLIB 动态代理。

CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB 是通过继承的方式做

的动态代理，因此如果某个类被标记为 `final`，那么它是无法使用 CGLIB 做动态代理的。

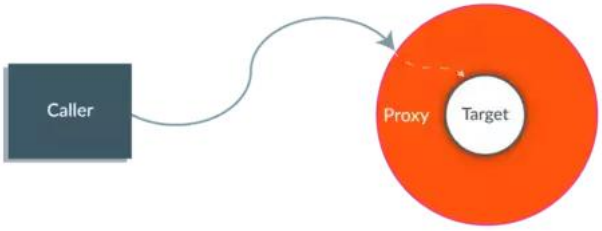
Spring AOP and AspectJ AOP 有什么区别?

- 代理方式不同
  - Spring AOP 基于动态代理方式实现。
  - AspectJ AOP 基于静态代理方式实现。
- PointCut 支持力度不同
  - Spring AOP **仅**支持方法级别的 PointCut 。
  - AspectJ AOP 提供了完全的 AOP 支持，它还支持属性级别的 PointCut 。

什么是编织（Weaving）？

Weaving ，**编织**。

- 为了创建一个 Advice 对象而链接一个 Aspect 和其它应用类型或对象，称为编织（Weaving）。
- 在 Spring AOP 中，编织在运行时执行，即动态代理。请参考下图：



Spring 如何使用 AOP 切面?

在 Spring AOP 中，有两种方式配置 AOP 切面：

- 基于 **XML** 方式的切面实现。
- 基于 **注解** 方式的切面实现。

目前，主流喜欢使用 **注解** 方式。胖友可以看看 《彻底征服 Spring AOP 之实战篇》 。

Spring Transaction

非常推荐阅读如下文章：

- 《可能是最漂亮的 Spring 事务管理详解》

什么是事务?

事务就是对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。

这样可以防止出现脏数据，防止数据库数据出现问题。

事务的特性指的是?

指的是 **ACID** ，如下图所示：

1. 原子性 Atomicity： 一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。

2. 一致性 Consistency： 在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器)、级联回滚等。

3. 隔离性 Isolation： 数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

4. 持久性 Durability ： 事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

列举 Spring 支持的事务管理类型?

目前 Spring 提供两种类型的事务管理：

- **声明式**事务：通过使用注解或基于 XML 的配置事务，从而事务管理与业务代码分离。
- **编程式**事务：通过编码的方式实现事务管理，需要在代码中显式的调用事务的获得、提交、回滚。它为您提供极大的灵活性，但维护起来非常困难。

实际场景下，我们一般使用 Spring Boot + 注解的**声明式**事务。具体的示例，胖友可以看看 《Spring Boot 事务注解详解》 。

另外，也推荐看看 《Spring 事务管理 – 编程式事务、声明式事务》 一文。

Spring 事务如何和不同的数据持久层框架做集成?

① 首先，我们先明确下，这里数据持久层框架，指的是 Spring JDBC、Hibernate、Spring JPA、MyBatis 等等。

② 然后，Spring 事务的管理，是通过 `org.springframework.transaction.PlatformTransactionManager` 进行管理，定义如下：

```
// PlatformTransactionManager.java

public interface PlatformTransactionManager {

    // 根据事务定义 TransactionDefinition ，获得 TransactionStatus
    TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;

    // 根据情况，提交事务
    void commit(TransactionStatus status) throws TransactionException;

    // 根据情况，回滚事务
    void rollback(TransactionStatus status) throws TransactionException;

}
```

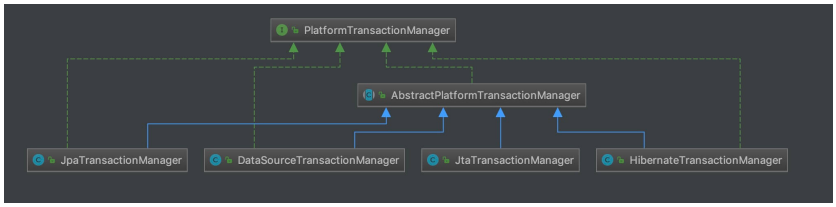
- PlatformTransactionManager 是负责事务管理的接口，一共有三个接口方法，分别负责事务的获得、提交、回滚。
- `#getTransaction(TransactionDefinition definition)` 方法，根据事务定义 TransactionDefinition ，获得

TransactionStatus 。
    - 为什么不是创建事务呢？因为如果当前如果已经有事务，则不会进行创建，一般来说会跟当前线程进行绑定。如果不存在事务，则进行创建。
    - 为什么返回的是 TransactionStatus 对象？在 TransactionStatus 中，不仅仅包含事务属性，还包含事务的其它信息，例如是否只读、是否为新创建的事务等等。😈 下面，也会详细解析 TransactionStatus 。
    - 事务 TransactionDefinition 是什么？😈 下面，也会详细解析 TransactionStatus 。
- `#commit(TransactionStatus status)` 方法，根据 TransactionStatus 情况，提交事务。
    - 为什么根据 TransactionStatus 情况，进行提交？例如说，带`@Transactional` 注解的的 A 方法，会调用 `@Transactional` 注解的的 B 方法。
        - 在 B 方法结束调用后，会执行 `PlatformTransactionManager#commit(TransactionStatus status)` 方法，此处事务**是不能、也不会**提交的。
        - 而是在 A 方法结束调用后，执行 `PlatformTransactionManager#commit(TransactionStatus status)` 方法，提交事务。
- `#rollback(TransactionStatus status)` 方法，根据 TransactionStatus 情况，回滚事务。
    - 为什么根据 TransactionStatus 情况，进行回滚？原因同 `#commit(TransactionStatus status)` 方法。

③ 再之后，PlatformTransactionManager 有**抽象子**类 `org.springframework.transaction.support.AbstractPlatformTransactionManager` ，基于 模板方法模式 ，实现事务整体逻辑的骨架，而抽象 `#doCommit(DefaultTransactionStatus status)`、`#doRollback(DefaultTransactionStatus status)` 等等方法，交由子类类来实现。

前方高能，即将进入关键的 ④ 步骤。

④ 最后，不同的数据持久层框架，会有其对应的 PlatformTransactionManager 实现类，如下图所示：



- 所有的实现类，都基于 AbstractPlatformTransactionManager 这个骨架类。
- HibernateTransactionManager ，和 Hibernate5 的事务管理做集成。
- DataSourceTransactionManager ，和 JDBC 的事务管理做集成。所以，它也适用于 MyBatis、Spring JDBC 等等。
- JpaTransactionManager ，和 JPA 的事务管理做集成。

如下，是一个比较常见的 XML 方式来配置的事务管理器，使用的是 DataSourceTransactionManager 。代码如下：

```
<!-- 事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 数据源 -->
    <property name="dataSource" ref="dataSource" />
</bean>
```

- 正如上文所说，它适用于 MyBatis、Spring JDBC 等等。

😈 是不是很有趣，更多详细的解析，可见如下几篇文章：

- 《精尽 Spring 源码分析 —— Transaction 源码简单导读》
- 《精尽 MyBatis 源码分析 —— 事务模块》

- 《精尽 MyBatis 源码解析 —— Spring 集成（四）之事务》

为什么在 Spring 事务中不能切换数据源?

做过 Spring 多数据源的胖友，都会有个惨痛的经历，为什么在开启事务的 Service 层的方法中，无法切换数据源呢？因为，在 Spring 的事务管理中，**所使用的数据库连接会和当前线程所绑定**，即使我们设置了另外一个数据源，使用的还是当前的数据源连接。

另外，多个数据源且需要事务的场景，本身会带来**多事务一致性**的问题，暂时没有特别好的解决方案。

所以一般一个应用，推荐除非了读写分离所带来的多数据源，其它情况下，建议只有一个数据源。并且，随着微服务日益身形，一个服务对应一个 DB 是比较常见的架构选择。

@Transactional 注解有哪些属性？如何使用?

`@Transactional` 注解的**属性**如下：

属性	类型
value	String
propagation	enum: Propagation
isolation	enum: Isolation
readOnly	boolean
timeout	int (in seconds granularity)
rollbackFor	Class对象数组，必须继承自Throwable
rollbackForClassName	类名数组，必须继承自Throwable
noRollbackFor	Class对象数组，必须继承自Throwable
noRollbackForClassName	类名数组，必须继承自Throwable

- 一般情况下，我们直接使用 `@Transactional` 的所有属性默认值即可。

具体**用法**如下：

- `@Transactional` 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，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。
- 虽然 `@Transactional` 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外，**`@Transactional` 注解应该只被应用到 `public` 方法上，这是由 Spring AOP 的本质决定的**。如果你在 `protected`、`private` 或者默认可见性的方法上使用 `@Transactional` 注解，这将被忽略，也不会抛出任何异常。**这一点，非常需要注意**。

下面，我们来简单说下**源码**相关的东西。

`@Transactional` 注解的属性，会解析成 `org.springframework.transaction.TransactionDefinition` 对象，即事务定义。TransactionDefinition 代码如下：

```
public interface TransactionDefinition {

    int getPropagationBehavior(); // 事务的传播行为
    int getIsolationLevel(); // 事务的隔离级别
    int getTimeout(); // 事务的超时时间
    boolean isReadOnly(); // 事务是否只读

    @Nullable
    String getName(); // 事务的名字

}
```

- 可能会胖友有以后，`@Transactional` 注解的 `rollbackFor`、`rollbackForClassName`、`noRollbackFor`、`noRollbackForClassName` 属性貌似没体现出来？它们提现在 TransactionDefinition 的实现类 RuleBasedTransactionAttribute 中。
- `#getPropagationBehavior()` 方法，返回事务的**传播行为**，该值是个枚举，在下面来说。
- `#getIsolationLevel()` 方法，返回事务的**隔离级别**，该值是个枚举，在下面来说。

什么是事务的隔离级别？分成哪些隔离级别？

关于这个问题，涉及的内容会比较多，胖友直接看如下两篇文章：

- 《数据库四大特性以及事务隔离级别》
- 《五分钟搞清楚 MySQL 事务隔离级别》

另外，有一点非常重要，不同数据库对四个隔离级别的支持和实现略有不同。因为我们目前互联网主要使用 MySQL 为主，所以至少要搞懂 MySQL 对隔离级别的支持和实现情况。

在 TransactionDefinition 接口中，定义了“**四种**”的隔离级别枚举。代码如下：

```
// TransactionDefinition.java

/**
 * 【Spring 独有】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
 */
int
```

```
ISOLATION_DEFAULT = -1;

/**
 * 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
 */
int ISOLATION_READ_UNCOMMITTED = Connection.TRANSACTION_READ_UNCOMMITTED;

/**
 * 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
 */
int ISOLATION_READ_COMMITTED = Connection.TRANSACTION_READ_COMMITTED;
/**
 * 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。
 */
int ISOLATION_REPEATABLE_READ = Connection.TRANSACTION_REPEATABLE_READ;
/**
 * 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。
     */
    int ISOLATION_SERIALIZABLE = Connection.TRANSACTION_SERIALIZABLE;
```

什么是事务的传播级别？分成哪些传播级别？

事务的**传播行为**，指的是当前带有事务配置的方法，需要怎么处理事务。

- 例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。
- 有一点需要注意，事务的传播级别，并不是数据库事务规范中的名词，**而是 Spring 自身所定义的**。通过事务的传播级别，Spring 才知道如何处理事务，是创建一个新事务呢，还是继续使用当前的事务。

  芳芳的自我吐槽：是不是有种背概念背的想哭

在 TransactionDefinition 接口中，定义了**三类七种**传播级别。代码如下：

```
// TransactionDefinition.java

// ========== 支持当前事务的情况 ==========

/**
     * 如果当前存在事务，则使用该事务。
     * 如果当前没有事务，则创建一个新的事务。
     */
    int PROPAGATION_REQUIRED = 0;
/**
     * 如果当前存在事务，则使用该事务。
     * 如果当前没有事务，则以非事务的方式继续运行。
     */
    int PROPAGATION_SUPPORTS = 1;
/**
     * 如果当前存在事务，则使用该事务。
     * 如果当前没有事务，则抛出异常。
     */
    int PROPAGATION_MANDATORY = 2;
```

```
    // ========= 不支持当前事务的情况 =========

    /**
     * 创建一个新的事务。
     * 如果当前存在事务，则把当前事务挂起。
     */
    int PROPAGATION_REQUIRES_NEW = 3;
    /**
     * 以非事务方式运行。
     * 如果当前存在事务，则把当前事务挂起。
     */
    int PROPAGATION_NOT_SUPPORTED = 4;
    /**
     * 以非事务方式运行。
     * 如果当前存在事务，则抛出异常。
     */
    int PROPAGATION_NEVER = 5;

    // ========= 其他情况 =========

    /**
     * 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，则等价于 {@link TransactionDefinition#PROPAGATION_REQUIRED}
     */
    int PROPAGATION_NESTED = 6;
```

- 分类之后，其实还是比较好记的。当然，绝大数场景，我们只用 `PROPAGATION_REQUIRED` 传播级别。
- 这里需要指出的是，前面的六种事务传播行为是 Spring 从 EJB 中引入的，他们共享相同的概念。而 `PROPAGATION_NESTED` 是

Spring 所特有的。

- 以 `PROPAGATION_NESTED` 启动的事务内嵌于外部事务中（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。如果熟悉 JDBC 中的保存点（SavePoint）的概念，那嵌套事务就很容易理解了，其实嵌套的子事务就是保存点的一个应用，一个事务中可以包括多个保存点，每一个嵌套子事务。另外，外部事务的回滚也会导致嵌套子事务的回滚。
- 😈 当然，虽然上面 `PROPAGATION_NESTED` 文字很长，实际我们基本没用过。或者说，去掉基本，我们根本没用过。

什么是事务的超时属性?

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。

在 TransactionDefinition 中以 `int` 的值来表示超时时间，其单位是秒。

当然，这个属性，貌似我们基本也没用过。

什么是事务的只读属性?

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。

- 所谓事务性资源就是指那些被事务管理的资源，比如数据源、JMS 资源，以及自定义的事务性资源等等。
- 如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。感兴趣的胖友，可以看看 《不使用事务和使用只读事务的区别》 。

在 TransactionDefinition 中以 `boolean` 类型来表示该事务是否只读。

什么是事务的回滚规则?

回滚规则，定义了哪些异常会导致事务回滚而哪些不会。

- 默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。
- 但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

注意，事务的回滚规则，并不是数据库事务规范中的名词，**而是 Spring 自身所定义的**。

简单介绍 TransactionStatus ?

芳芳：这个可能不是一个面试题，主要满足下大家的好奇心。

TransactionStatus 接口，记录事务的状态，不仅仅包含事务本身，还包含事务的其它信息。代码如下：

```
// TransactionStatus.java

public interface TransactionStatus extends SavepointManager, Flushable {

    /**
     * 是否是新创建的事务
     */
    boolean isNewTransaction();

    /**
     * 是否有 Savepoint
     * 在 {@link TransactionDefinition#PROPAGATION_NESTED} 传播级别使用。
     */
    boolean hasSavepoint();

    /**
     * 设置为只回滚
     */
    void setRollbackOnly();

    /**
     * 是否为只回滚
     */
    boolean isRollbackOnly();

    /**
     * 执行 flush 操作
     */
    @Override
    void flush();

    /**
     * 是否事务已经完成
     */
    boolean isCompleted();

}
```

- 为什么没有事务对象呢？在 TransactionStatus 的实现类 DefaultTransactionStatus 中，有个 `Object transaction` 属性，表示事务对象。

- `#isNewTransaction()` 方法，表示是否是新创建的事务。有什么用呢？答案结合 「Spring 事务如何和不同的数据持久层框架做集成？」 问题，我们对 `#commit(TransactionStatus status)` 方法的解释。通过该方法，我们可以判断，当前事务是否当前方法所创建的，只有创建事务的方法，**才能且应该真正的提交事务**。

使用 Spring 事务有什么优点?

1. 通过 PlatformTransactionManager ，为不同的数据层持久框架提供统一的 API ，无需关心到底是原生 JDBC、Spring JDBC、JPA、Hibernate 还是 MyBatis 。
2. 通过使用声明式事务，使业务代码和事务管理的逻辑分离，更加清晰。

从倾向上来说，芳芳比较喜欢**注解** + 声明式事务。

Spring Data Access

芳芳：这块的问题，感觉面试问的不多，至少我很少问。哈哈哈。就当做下了解，万一问了呢。

Spring 支持哪些 ORM 框架?

- Hibernate
- JPA
- MyBatis
- JDO
- OJB

可能会有胖友说，不是应该还有 Spring JDBC 吗。注意，Spring JDBC 不是 ORM 框架。

在 Spring 框架中如何更有效地使用 JDBC ?

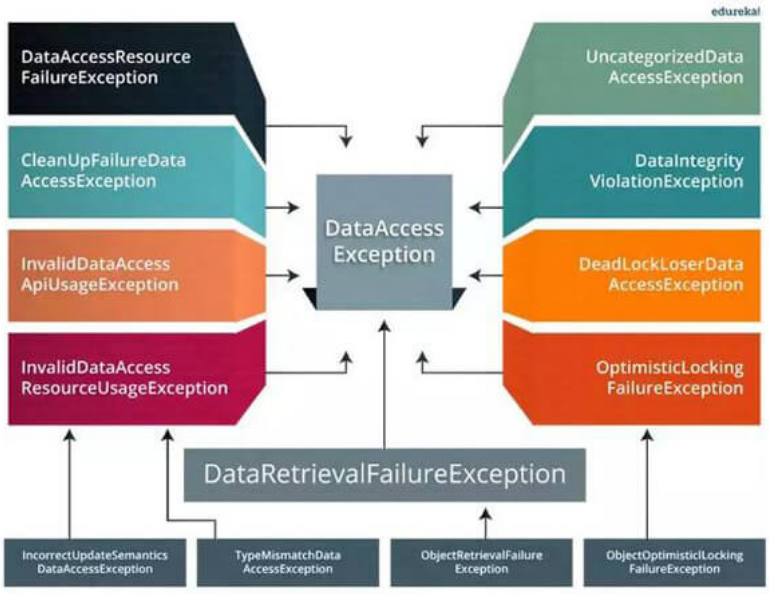
Spring 提供了 Spring JDBC 框架，方便我们使用 JDBC 。

对于开发者，只需要使用 JdbcTemplate 类，它提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

没有使用过的胖友，可以看看 《Spring JDBC 访问关系型数据库》 文章。

Spring 数据数据访问层有哪些异常?

通过使用 Spring 数据数据访问层，它统一了各个数据持久层框架的不同异常，统一进行提供 `org.springframework.dao.DataAccessException` 异常及其子类。如下图所示：



使用 Spring 访问 Hibernate 的方法有哪些?

芳芳：这个问题很灵异，因为芳芳已经好久不使用 Hibernate 了，所以答案是直接复制的。

我们可以通过两种方式使用 Spring 访问 Hibernate：

- 使用 Hibernate 模板和回调进行控制反转。
- 扩展 HibernateDAOSupport 并应用 AOP 拦截器节点。

芳芳：不过我记得，12 年我用过 Spring JPA 的方式，操作 Hibernate 。具体可参考 《一起来学 SpringBoot 2.x | 第六篇：整合 Spring Data JPA》 。

当然，我们可以再来看一道 《JPA 规范与 ORM 框架之间的关系是怎样的呢？》 。这个问题，我倒是问过面试的候选人，哈哈哈哈。

666. 彩蛋

整理 Spring 面试题的过程中，又把 Spring 的知识点又复习了一遍。我突然有点想念，那本被我翻烂的 《Spring 实战》 。

😈 我要买一本原版的！！！

参考与推荐如下文章：

- Java 架构 《Spring 面试题》
- 永顺 《彻底征服 Spring AOP 之理论篇》
- 陌上桑花开花 《Java 面试题集（七）– Spring常见面试问题》
- 一人浅醉 《Spring 的 @Transactional 注解详细用法》
- dalaoyang 《Spring 面试题》