@来源 Spring全家桶面试题（2021优化版）(qq.com)

Spring概述

@$什么是Spring?

Spring是**一个轻量级Java开源框架**，最早由Rod Johnson创建，目的是**解决企业级应用开发的复杂性，简化Java开发。**Spring为开发Java应用程序提供全面的基础架构支持，因此Java开发者可以专注于应用程序的开发。

Spring可以做很多事情，它为企业级开发提供给了丰富的功能，但是这些功能的底层都依赖于它的两个核心特性，也就是**依赖注入（dependency injection，DI）**和**面向切面编程（aspect-oriented programming，AOP）**。

为了降低Java开发的复杂性，Spring采取了以下4种关键策略

- 基于POJO的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

Spring框架的设计目标，设计理念，和核心是什么

Spring设计目标：Spring为开发者提供一个一站式轻量级应用开发平台；

Spring设计理念：在JavaEE开发中，支持POJO和JavaBean开发方式，使应用面向接口开发，充分支持OO（面向对象）设计方法；Spring通过IoC容器实现对象耦合关系的管理，并实现依赖反转，将对象之间的依赖关系交给IoC容器，实现解耦；

Spring框架的核心：IoC容器和AOP模块。通过IoC容器管理POJO对象以及他们之间的耦合关系；通过AOP以动态非侵入的方式增强服务，把遍布于应用各层的功能分离出来形成可重用的功能组件。

Spring的优缺点是什么?

优点

- 方便解耦，简化开发

  Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。

- AOP编程的支持

  Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。

- 声明式事务的支持

  只需要通过配置就可以完成对事务的管理，而无需手动编程。

- 方便程序的测试

  Spring对Junit4支持，可以通过注解方便的测试Spring程序。

- 方便集成各种优秀框架

  Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。

- 降低JavaEE API的使用难度

  Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

缺点

- Spring明明一个很轻量级的框架，却给人感觉大而全
- Spring依赖反射，反射影响性能
- 使用门槛升高，入门Spring需要较长时间

Spring有哪些应用场景

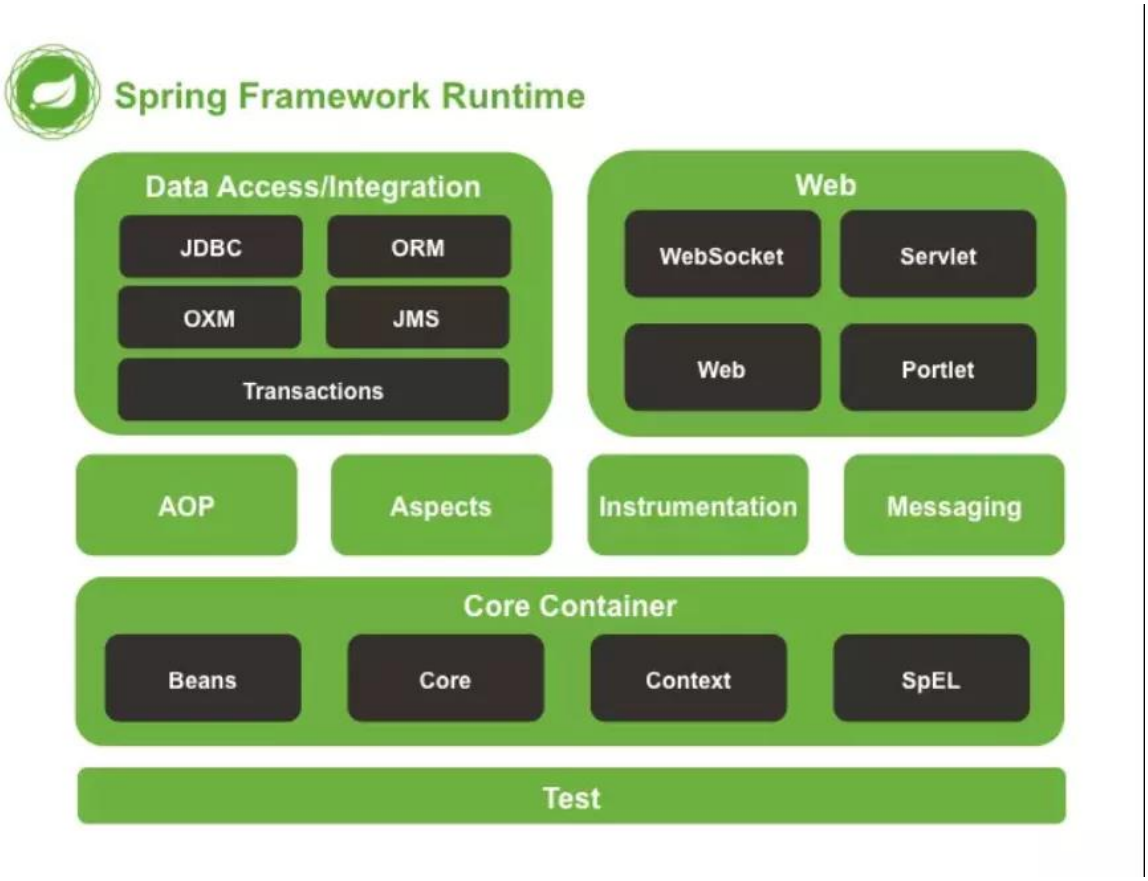
应用场景：JavaEE企业应用开发，包括SSH、SSM等

Spring价值：

- Spring是非侵入式的框架，目标是使应用程序代码对框架依赖最小化；
- Spring提供一个一致的编程模型，使应用直接使用POJO开发，与运行环境隔离开来；
- Spring推动应用设计风格向面向对象和面向接口开发转变，提高了代码的重用性和可测试性；

Spring由哪些模块组成?

Spring 总共大约有 20 个模块， 由 1300 多个不同的文件构成。而这些组件被分别整合在核心容器（Core Container） 、 AOP（Aspect Oriented Programming）和设备支持（Instrumentation） 、数据访问与集成（Data Access/Integration） 、 Web、 消息（Messaging） 、 Test等 6 个模块中。以下是 Spring 5 的模块结构图：



- spring core：提供了框架的基本组成部分，包括控制反转（Inversion of Control，IoC）和依赖注入（Dependency Injection，DI）功能。
- spring beans：提供了BeanFactory，是工厂模式的一个经典实现，Spring将管理对象称为Bean。
- spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
- spring jdbc：提供了一个JDBC的抽象层，消除了烦琐的JDBC编码和数据库厂商特有的错误代码解析， 用于简化JDBC。
- spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
- spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。

- spring test：主要为测试提供支持的，支持使用JUnit或TestNG对Spring组件进行单元测试和集成测试。

@$Spring 框架中都用到了哪些设计模式?

1. 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
2. 单例模式：Bean默认为单例模式。
3. 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
4. 模板方法：用来解决代码重复的问题。比如：RestTemplate, JmsTemplate, JpaTemplate。
5. 观察者模式：定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被动更新，如Spring中listener的实现--ApplicationListener。

控制反转(IoC)

@$什么是Spring IoC 容器?

控制反转即IoC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"概念就是**对对象组件控制权的转移，从程序代码本身转移到了外部容器。**

Spring IoC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。）

@$控制反转(IoC)有什么作用

- **管理对象的创建和依赖关系的维护。**对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
- **解耦，由容器去维护具体的对象**
- **托管了类的整个生命周期**，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

Spring IoC 的实现机制

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

示例：

```
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 {
```

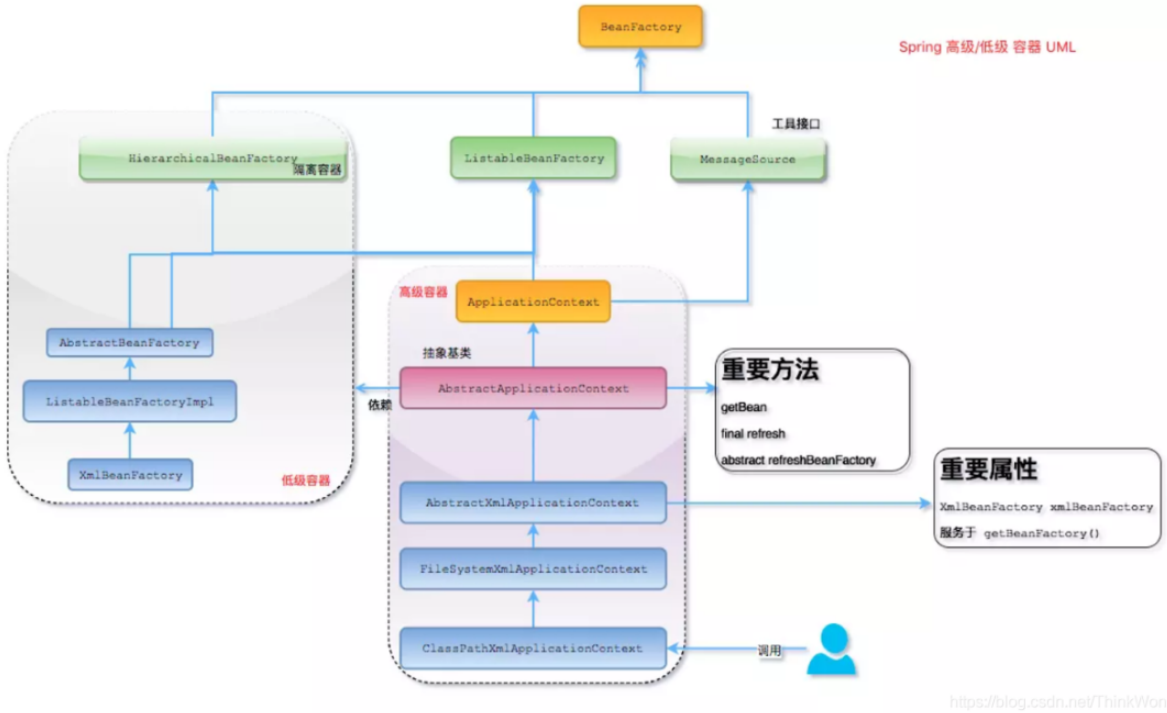
```
            f=(Fruit)Class.forName(ClassName).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
         return f;
    }
}

class Client {
    public static void main(String[] a) {
        Fruit f=Factory.getInstance("com.jourwon.spring.Apple");
        if(f!=null){
            f.eat();
        }
    }
}
```

BeanFactory 和 ApplicationContext有什么区别?

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的 ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。



依赖关系

BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。我们可以称之为 “**低级容器**”。

ApplicationContext接口作为BeanFactory的派生，可以称之为 “**高级容器**”。除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。
- 统一的资源文件访问方式。
- 提供在监听器中注册bean的事件。
- 同时加载多个配置文件。

- 载入多个（有继承关系）上下文，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

加载方式

BeanFactroy采用的是**延迟加载**形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，**一次性创建了所有的Bean**。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean，确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

创建方式

BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

注册方式

BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要**手动注册**，而ApplicationContext则是**自动注册**。

@$什么是Spring的依赖注入(Dependency Injection)?

控制反转IoC是一个很大的概念，可以有不同的实现方式。其主要实现方式有两种：依赖注入和依赖查找

依赖注入：相对于IoC而言，依赖注入(DI)更加准确地描述了IoC的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是**由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的Java方法，让容器去决定依赖关系。**

依赖查找（Dependency Lookup）：容器提供回调接口和上下文环境给组件。EJB和Apache Avalon都使用这种方式。

依赖查找也有两种类型：依赖拖拽（DP）和上下文依赖查找(CDL)。

有哪些不同类型的依赖注入实现方式?

依赖注入是时下最流行的IoC实现方式，依赖注入分为接口注入（Interface Injection），Setter方法注入（Setter Injection）和构造器注入（Constructor Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从Spring4开始已被废弃。

构造器注入：构造器注入是容器通过调用一个类的构造器来实现的，该构造器有一系列参数，每个参数都必须注入。

Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂方法实例化bean之后，调用该bean的setter方法来实现的依赖注入。

@$构造器注入和 Setter方法注入的区别

	构造器注入	setter 注入
部分注入	没有部分注入	有部分注入
覆盖setter属性	不会覆盖 setter 属性	会覆盖 setter 属性
任意修改是否创建新实例	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
适用场景	适用于设置很多属性	适用于设置少量属性

最好的解决方案是用**构造器参数实现强制依赖，setter方法实现可选依赖。**

面向切面编程(AOP)

@$什么是AOP

OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并不适用于定义横向的关系，导致了大量重复代码，而不利于各个模块的重用。

AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），通过面向切面编程减少了系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。常用于权限认证、日志、事务处理等。

Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式?

AOP实现的关键在于代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的**动态代理**，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，**这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。**

JDK动态代理和CGLIB动态代理的区别

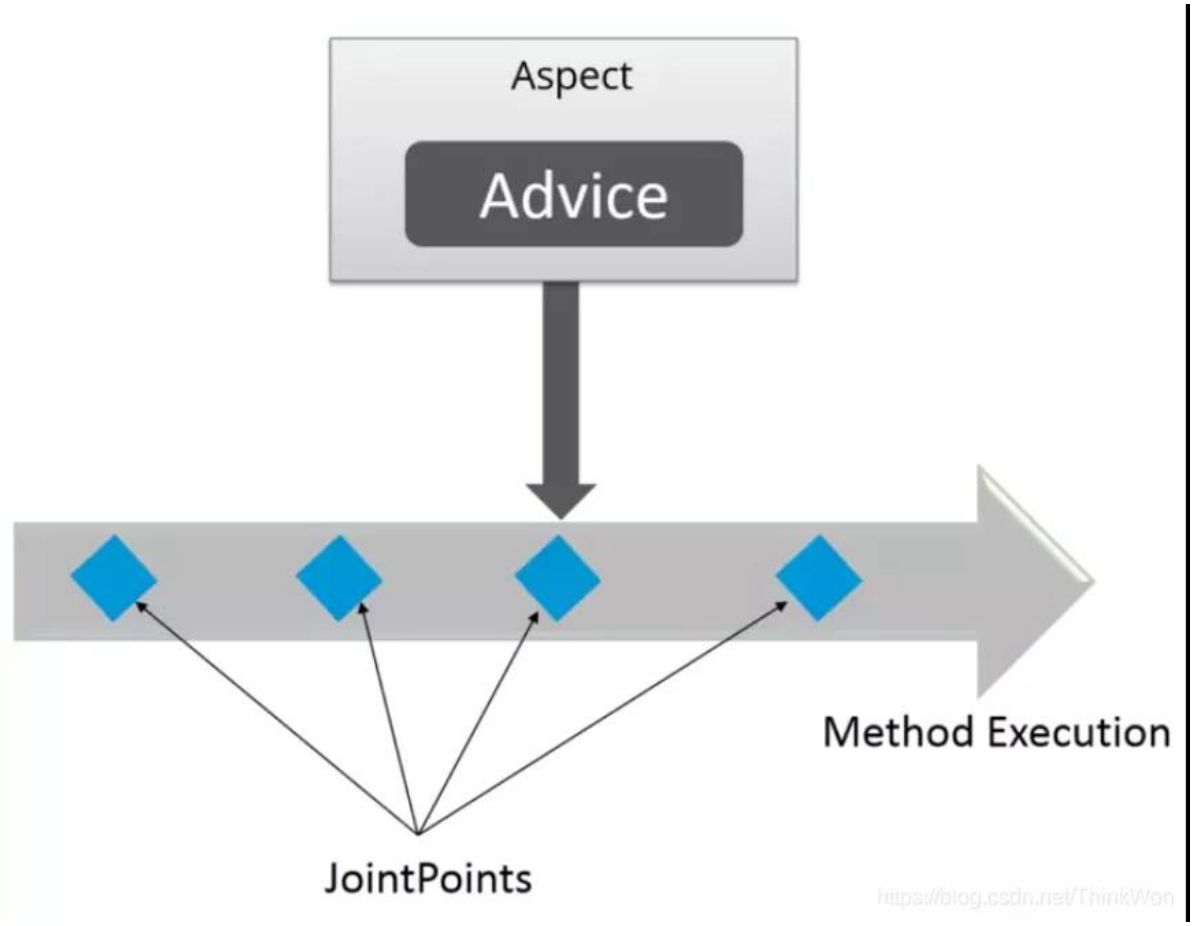
Spring AOP中的动态代理主要有两种方式，**JDK动态代理**和**CGLIB动态代理**:

- JDK动态代理**只提供接口的代理**，不支持类的代理。核心InvocationHandler接口和Proxy类，**InvocationHandler 通过invoke()方法反射来调用目标类中的代码**，动态地将横切逻辑和业务编织在一起；接着，**Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例，生成目标类的代理对象。**
- 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。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指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。**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。

静态代理与动态代理区别在于**生成AOP代理对象的时机不同**，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

> InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

解释一下Spring AOP里面的几个名词



（1）切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）通知（Advice）：在AOP术语中，切面的工作被称为通知。

（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）目标对象（Target Object）：被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。也有人把它叫做 被通知（adviced） 对象。既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

Spring通知有哪些类型?

在AOP术语中，切面的工作被称为通知，实际上是程序执行时要通过SpringAOP框架触发的代码段。

以下是Spring定义的五种通知和常见使用场景

通知类型	说明	使用场景
前置通知（Before）	在目标方法被执行之前调用通知	
后置通知（After）	无论如何都会在目标方法执行之后调用通知	记录日志(方法已经调用，但不一定成功)
返回通知（After-returning）	在目标方法正常返回后执行通知，如果方法没有正常返返回，例如抛出异常，则返回通知不会执行，可以通过配置得到目标方法的返回值	记录日志(方法已经成功调用)
异常通知（After-throwing）	在目标方法抛出异常后调用通知	异常处理
环绕通知（Around）	通知包裹了目标方法，在目标方法调用之前和调用之后执行自定义的行为	事务权限控制

Spring Beans

解释Spring支持的几种bean的作用域

当定义一个bean在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean的scope属性来定义。

Spring框架支持以下五种bean的作用域：

作用域	描述
singleton	单例模式，在spring IoC容器仅存在一个Bean实例，默认值
prototype	原型模式，每次从容器中获取Bean时，都返回一个新的实例，即每次调用getBean()时，相当于执行newXxxBean()
request	每次HTTP请求都会创建一个新的Bean，该作用域仅在基于web的Spring ApplicationContext环境下有效
session	同一个HTTP Session共享一个Bean，不同Session使用不同的Bean，该作用域仅在基于web的Spring ApplicationContext环境下有效
global-session	同一个全局的HTTP Session中共享一个Bean，一般用于Portlet应用环境，该作用域仅在基于web的Spring ApplicationContext环境下有效

注意： 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

Spring框架中的单例bean是线程安全的吗?

不是，Spring框架中的单例bean不是线程安全的。

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 是无状态的（比如 dao 类），所以某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。

有状态和无状态区别

- 有状态就是有数据存储功能。
- 无状态就是不会保存数据。

@$Spring如何处理线程并发问题?

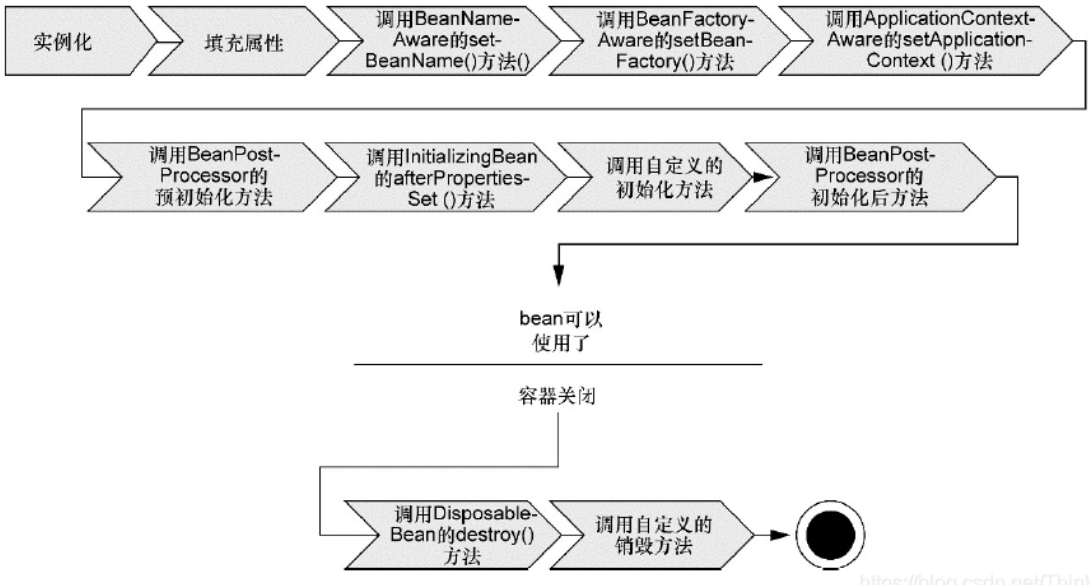
在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中共享变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

@$解释Spring框架中bean的生命周期

在传统的Java应用中，bean的生命周期很简单。使用Java关键字new进行bean实例化，然后该bean就可以使用了。一旦该bean不再被使用，则由Java自动进行垃圾回收。相比之下，Spring容器中的bean的生命周期就显得相对复杂多了。正确理解Spring bean的生命周期非常重要，因为你或许要利用Spring提供的扩展点来自定义bean的创建过程。下图展示了bean装载到Spring应用上下文中的一个典型的生命周期过程。



bean在Spring容器中从创建到销毁经历了若干阶段，每一阶段都可以针对Spring如何管理bean进行个性化定制。

正如你所见，在bean准备就绪之前，bean工厂执行了若干启动步骤。

我们对上图进行详细描述：

Spring对bean进行实例化；

Spring将值和bean的引用注入到bean对应的属性中；

如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBeanName()方法；

如果bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方法，将BeanFactory容器实例传入；

如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext()方法，将bean所在的应用上下文的引用传入进来；

如果bean实现了BeanPostProcessor接口，Spring将调用它们的postProcessBeforeInitialization()方法；

如果bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet()方法。类似地，如果bean使用initmethod声明了初始化方法，该方法也会被调用；

此时，bean已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；

如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方法。同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。

现在你已经了解了如何创建和加载一个Spring容器。但是一个空的容器并没有太大的价值，在你把东西放进去之前，它里面什么都没有。为了从Spring的DI(依赖注入)中受益，我们必须将应用对象装配进Spring容器中。

Spring注解

使用@Autowired注解自动装配bean的过程是怎样的?

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置`<context:annotation-config />`标签，然后在启动spring IoC时，容器就会**自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器**，当容器扫描到**@Autowied**、**@Resource**或**@Inject**时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

- 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的属性；
- 如果查询的结果不止一个，会抛出异常，需要配合**@Qualifier注解根据名称来查找**；
- 如果配合@Qualifier注解根据名称来查找的结果为空，会抛出异常，可以将@Autowire注解的required属性设置为false。

@Autowired和@Resource之间的区别

@Autowired和@Resource都可用于：构造函数、Setter方法、成员变量 ，都可以用来装配bean

@Autowired和@Resource之间的区别

- @Autowired默认**按类型装配**（这个注解是属于spring的），@Resource 是JDK1.6支持的注解，**默认按照名称进行装配**
- @Autowired**默认情况下必须要求依赖对象必须存在**，如果**要允许null值**，可以**设置它的required属性为false**，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用

```
@Autowired
@Qualifier("baseDao")
private BaseDao baseDao;
```

Resource名称可以**通过name属性进行指定**，如果没有指定name属性，当注解写在字段上时，**默认取字段名**，按照名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

```
@Resource(name="baseDao")
private BaseDao baseDao;
```

- 其实@Autowired + @Qualifier("BWM") == @Resource(name="BWM")，如果当一个接口只有一个实现类，使用@Autowired和@Resource没有区别，但是最好用@Autowired

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

这四个注解都可以将bean注入到spring容器中，根据不同的使用场景定义了特定功能的注解组件

@Controller用于标注控制层组件

@Service用于标注业务层组件

@Repository用于标注数据访问层组件，即DAO组件

@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注

数据访问

@$Spring支持的事务管理类型， Spring事务实现方式有哪些？你更倾向用哪种事务管理类型?

Spring支持两种类型的事务管理：

编程式事务管理：通过编程的方式管理事务，灵活性好，但是难维护。

声明式事务管理：将业务代码和事务管理分离，只需用注解和XML配置来管理事务。

大多数情况下选择声明式事务管理，虽然比编程式事务管理少了一点灵活性，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别，但是声明式事务管理对应用代码的影响最小，更符合一个无侵入的轻量级容器的思想，具有更好的可维护性。

Spring事务的实现方式和实现原理

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过bin log或者redo log实现的。

@$说一下Spring的事务传播行为

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① **PROPAGATION_REQUIRED**：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② **PROPAGATION_SUPPORTS**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ **PROPAGATION_MANDATORY**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ **PROPAGATION_REQUIRES_NEW**：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ **PROPAGATION_NOT_SUPPORTED**：以非事务方式执行，如果当前存在事务，就把当前事务挂起。

⑥ **PROPAGATION_NEVER**：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ **PROPAGATION_NESTED**：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

说一下 spring 的事务隔离级别?

spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1. ISOLATION_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
2. ISOLATION_READ_UNCOMMITTED：读未提交，最低的隔离级别，一个事务可以读取另一个事务更新但未提交的数据。（会出现脏读、不可重复读、幻读）；
3. ISOLATION_READ_COMMITTED：读已提交，一个事务提交后才能被其他事务读取到（会出现不可重复读、幻读），Oracle、SQL server 的默认级别；
4. ISOLATION_REPEATABLE_READ：可重复读，对同一字段的多次读取结果都是一致的，除非数据被本身事务所修改（会出现幻读），MySQL 的默认级别；
5. ISOLATION_SERIALIZABLE：可串行化，最高的隔离级别，可以防止脏读、不可重复读、幻读。

Spring框架的事务管理有哪些优点?

- 为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
- 为编程式事务管理提供了一套简单的API而不是一些复杂的事务API
- 支持声明式事务管理。
- 和Spring各种数据访问抽象层很好的集成。

Spring MVC

@$什么是Spring MVC？简单介绍下你对Spring MVC的理解?

Spring MVC是一个基于Java，实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把模型-视图-控制器分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

Spring MVC的优点

（1）可以支持各种视图技术，而不仅仅局限于JSP；

（2）与Spring框架集成（如IoC容器、AOP等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet) , 处理器映射器（handlerMapping），处理器适配器（HandlerAdapter），视图解析器（ViewResolver）。

（4） 支持各种请求资源的映射策略。

Spring MVC的主要组件?

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器HandlerMapping（不需要程序员开发）

作用：根据请求的URL来查找Handler

（3）处理器适配器HandlerAdapter

注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。

（4）处理器Handler（需要程序员开发）

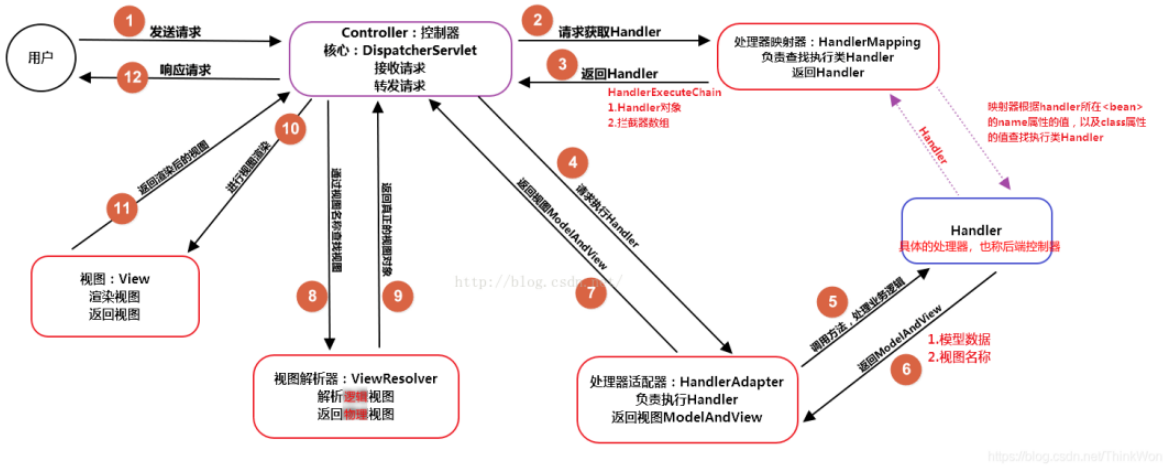
（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图View（需要程序员开发jsp）

View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

@$请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程?



（1）用户发送请求至前端控制器DispatcherServlet；

（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handler；

（3）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；

（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；

（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；

（6）Handler执行完成返回ModelAndView；

（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；

（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；

（9）ViewResolver解析后返回具体View；

（10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）

（11）DispatcherServlet响应用户。

MVC是什么？MVC设计模式的好处有哪些

mvc是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验的总结）。模型（model）-视图（view）-控制器（controller），三层架构的设计模式。用于实现前端页面的展现与后端业务数据处理的分离。

mvc设计模式的好处

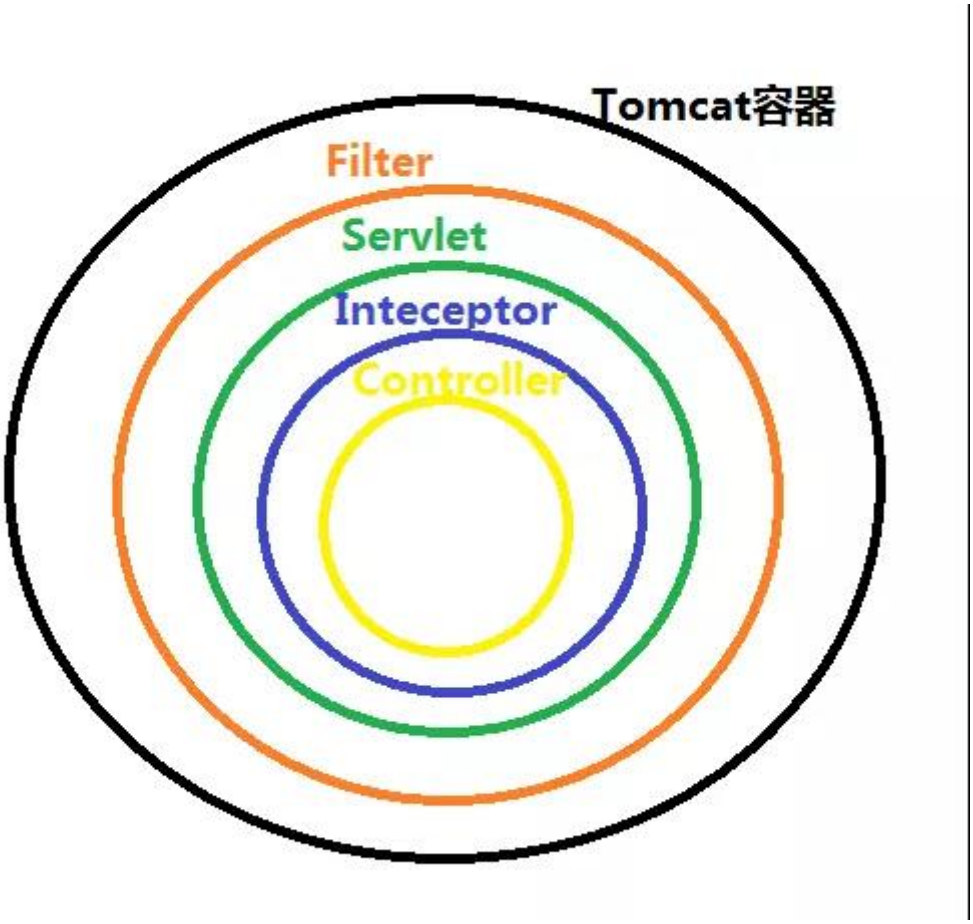
1. 分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。
2. 有利于系统的并行开发，提升开发效率。

@$拦截器Interceptor与过滤器Filter的区别

拦截器和过滤器的区别

类型	过滤器Filter	拦截器interceptor
规范	Filter是在Servlet规范中定义的，是Servlet容器支持的	拦截器是在Spring容器内的，是Spring框架支持的
使用范围	过滤器只能用于Web程序中	拦截器既可以用于Web程序，也可以用于Application、Swing程序中
原理	过滤器是基于函数回调	拦截器是基于java的反射机制
使用的资源	过滤器不能使用Spring资源	拦截器是一个Spring的组件，归Spring管理，配置在Spring文件中，因此能使用Spring里的任何资源、对象，例如Service对象、数据源、事务管理等，可以通过IoC注入到拦截器
深度	Filter在只在Servlet前后起作用	拦截器能够深入到方法前后、异常抛出前后等，因此拦截器的使用具有更大的弹性

在Tomcat容器中，过滤器和拦截器触发时机不一样，过滤器是在请求进入容器后，但请求进入servlet之前进行预处理的。请求结束返回也是，是在servlet处理完后，返回给前端之前。过滤器包裹住servlet，servlet包裹住拦截器



@$Spring MVC与Struts2区别

相同点

都是基于mvc设计模式的web框架，都用于web项目的开发。在企业项目中，Spring MVC使用更多一些。

不同点

类型	Spring MVC	Struts2
前端控制器	Spring MVC的前端控制器是servlet：DispatcherServlet	Struts2的前端控制器是filter：StrutsPreparedAndExcutorFilter
拦截机制和请求参数的接受方式	Spring MVC是方法级别的拦截，一个方法对应一个request上下文。Spring MVC是使用方法的形参接收请求的参数，基于方法的开发，线程安全，单例模式	Struts2框架是类级别的拦截，每次请求就会创建一个Action，一个Action对应一个request上下文。Struts2是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，多例模式
配置和性能	配置少，开发效率和性能高于Struts2	配置多，开发效率和性能低于Spring MVC
与Spring框架的整合	Spring MVC是Spring框架的一部分，无缝整合	Struts2与Spring整合相对麻烦

Spring Boot

@$什么是 Spring Boot?

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简化了繁重的配置，提供了各种启动器，开发者能快速上手。

Spring Boot 有哪些优点?

Spring Boot 主要有如下优点：

1. 容易上手，提升开发效率，为 Spring 开发提供一个更快、更广泛的入门体验。
2. 开箱即用，远离繁琐的配置。
3. 没有代码生成，也不需要XML配置。
4. 提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
5. 避免大量的 Maven 导入和各种版本冲突。

@$Spring Boot的启动流程

Spring Boot 入口——main方法

```
@SpringBootApplication
public class Application {
   public static void main(String[] args) throws Exception {
       SpringApplication.run(Application.class, args);
  }
}
```

从上面代码可以看出，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，所以分析 Spring Boot 启动过程，我们就从这两方面开始。

从源码声明可以看出，**@SpringBootApplication相当于 @SpringBootConfiguration + @ComponentScan + @EnableAutoConfiguration**，因此我们直接拆开来分析。

上面三个注解都在做一件事：**注册bean到spring容器。**他们通过不同的条件不同的方式来完成：

- @SpringBootConfiguration 通过与 @Bean 结合完成Bean的 JavaConfig 配置；
- @ComponentScan 通过范围扫描的方式，扫描特定注解注释的类，将其注册到Spring容器；
- @EnableAutoConfiguration 通过 spring.factories 的配置，并结合 @Condition 条件，完成bean的注册；

除了上面的三个注解，还可以使用@Import注解将bean注册到Spring容器

- @Import 通过导入的方式，将指定的class注册解析到Spring容器；

Spring Boot 启动流程

SpringApplication的实例化

- 推断应用类型是否是Web环境
- 设置初始化器（Initializer）
- 设置监听器（Listener）
- 推断应用入口类（Main）

SpringApplication.run方法

- 获取SpringApplicationRunListeners
- 准备配置环境ConfigurableEnvironment
- 创建ApplicationContext应用上下文
- ApplicationContext前置处理
- ApplicationContext刷新
- ApplicationContext后置处理

完成了实例化，下面开始调用run方法

```
// 运行run方法
public ConfigurableApplicationContext run(String... args) {
   // 此类通常用于监控开发过程中的性能，而不是生产应用程序的一部分。
   StopWatch stopWatch = new StopWatch();
   stopWatch.start();

   ConfigurableApplicationContext context = null;
   Collection<SpringBootExceptionReporter> exceptionReporters = new
ArrayList<>();

   // 设置java.awt.headless系统属性，默认为true
   // Headless模式是系统的一种配置模式。在该模式下，系统缺少了显示设备、键盘或鼠标。
   configureHeadlessProperty();

   // KEY 1 - 获取SpringApplicationRunListeners
   SpringApplicationRunListeners listeners = getRunListeners(args);

   // 通知监听者，开始启动
   listeners.starting();
   try {
       ApplicationArguments applicationArguments = new
DefaultApplicationArguments(
               args);

       // KEY 2 - 根据SpringApplicationRunListeners以及参数来准备环境
       ConfigurableEnvironment environment = prepareEnvironment(listeners,
               applicationArguments);

       configureIgnoreBeanInfo(environment);

       // 准备Banner打印器 - 就是启动Spring Boot的时候打印在console上的ASCII艺术字
体
       Banner printedBanner = printBanner(environment);

       // KEY 3 - 创建Spring上下文
       context = createApplicationContext();

       // 注册异常分析器
       analyzers = new FailureAnalyzers(context);

       // KEY 4 - Spring上下文前置处理
       prepareContext(context, environment, listeners, applicationArguments,
               printedBanner);

       // KEY 5 - Spring上下文刷新
       refreshContext(context);

       // KEY 6 - Spring上下文后置处理
```

```
        afterRefresh(context, applicationArguments);

        // 发出结束执行的事件
        listeners.finished(context, null);

        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                    .logStarted(getApplicationLog(), stopWatch);
       }
        return context;
   }
    catch (Throwable ex) {
        handleRunFailure(context, listeners, exceptionReporters, ex);
        throw new IllegalStateException(ex);
   }
}
```

@$Spring Boot Starter是什么？如何自定义Spring Boot Starter

Spring boot之所以流行，很大原因是因为有Spring Boot Starter。Spring Boot Starter是Spring boot的核心，可以理解为一个可拔插式的插件，例如，你想使用Reids插件，那么可以导入spring-boot-starter-redis依赖

Starter的命名

官方对Starter项目的jar包定义的 artifactId 是有要求的，当然也可以不遵守。Spring官方Starter通常命名为spring-boot-starter-{name}如：spring-boot-starter-web，Spring官方建议非官方的starter命名应遵守{name}-spring-boot-starter的格式。

传统的做法

在没有starter之前，假如我想要在Spring中使用jpa，那我可能需要做以下操作：

1. 在Maven中引入使用的数据库的依赖（即JDBC的jar）
2. 引入jpa的依赖
3. 在xxx.xml中配置一些属性信息
4. 反复的调试直到可以正常运行

需要注意的是，这里操作在我们**每次新建一个需要用到jpa的项目都需要重复的做一次。**

使用Spring Boot Starter可以提升效率

starter的主要目的就是为了解决上面的这些问题。

使用starter的好处，starter的作用：

- 帮助用户去除了繁琐的重复性的构建操作
- 在“约定大于配置”的理念下，ConfigurationProperties还帮助用户减少了无谓的配置操作
- 因为 application.properties 文件的存在，用户可以集中管理自定义配置

创建自己的Spring Boot Starter

如果你想要自己创建一个starter，那么基本上包含以下几步

1. 新建一个Maven项目，在pom.xml文件中定义好所需依赖；
2. 新建配置类，写好配置项和默认值，使用@ConfigurationProperties指明配置项前缀；
3. 新建自动装配类，使用@Configuration和@Bean来进行自动装配；
4. 新建spring.factories文件，用于指定自动装配类的路径；
5. 将starter安装到maven仓库，让其他项目能够引用

> spring.factories文件位于resources/META-INF目录下，需要手动创建;`org.springframework.boot.autoconfigure.EnableAutoConfiguration`后面的类名说明了自动装配类，如果有多个，则用逗号分开；使用者应用（SpringBoot）在启动的时候，会通过`org.springframework.core.io.support.SpringFactoriesLoader`读取classpath下每个Starter的spring.factories文件，加载自动装配类进行Bean的自动装配；

什么是 JavaConfig?

Spring JavaConfig 是 Spring 社区的产品，它提供了使用注释来配置Bean的纯 Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

（1）面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

（2）减少或消除 XML 配置。基于依赖注入原则的外部配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将JavaConfig 与 XML 混合匹配是理想的。

（3）类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

Spring boot 核心配置文件是什么？bootstrap.properties 和 application.properties 有何区别？

单纯做 Spring Boot 开发，不太容易遇到 bootstrap.properties 配置文件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时侯。

spring boot 核心的两个配置文件：

- bootstrap (. yml 或者 . properties)：bootstrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。且 bootstrap 里面的属性不能被覆盖；
- application (. yml 或者 . properties)：由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

什么是 Spring Profiles?

Spring Profiles 主要有下面两个使用场景：

- 根据不同的使用环境定义不同的profiles文件，比如开发，测试和生产，可以大大省去我们修改配置信息而带来的烦恼
- 根据不同的profiles（dev，test，prod）注册不同的bean，当应用程序在开发中运行时，只有某些 bean 可以加载，而在生产中，某些其他 bean 可以加载

spring-boot-starter-parent 有什么用？

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent ，spring-boot-starter-parent 主要有如下作用：

1. 定义了 Java 编译版本为 1.8 。
2. 使用 UTF-8 格式编码。
3. 继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
4. 执行打包操作的配置。
5. 自动化的资源过滤。
6. 自动化的插件配置。

7. 针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

Spring Boot 打成的 jar 和普通的 jar 有什么区别？

Spring Boot 项目最终打包成的 jar 是可执行 jar，这种 jar 可以直接通过 `java -jar xxx.jar` 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 `\BOOT-INF\classes` 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar，一个可执行，一个可引用。

Spring Boot 中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过 （CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS，现在可以通过实现 WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题。

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

   @Override
   public void addCorsMappings(CorsRegistry registry) {
       registry.addMapping("/**")
               .allowedOrigins("*")
               .allowCredentials(true)
               .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
               .maxAge(3600);
  }

}
```

使用上面的方式解决跨域会有点问题，具体描述如下

我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。

当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。

我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

```
@Configuration
public class CorsConfig {

   @Bean
   public CorsFilter corsFilter() {
       CorsConfiguration corsConfiguration = new CorsConfiguration();
       corsConfiguration.addAllowedOrigin("*");
       corsConfiguration.addAllowedHeader("*");
       corsConfiguration.addAllowedMethod("*");
       corsConfiguration.setAllowCredentials(true);
```

```
        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new UrlBasedCorsConfigurationSource();
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**", corsConfiguration);
        return new CorsFilter(urlBasedCorsConfigurationSource);
    }

}
```

Spring Cloud

为什么需要学习Spring Cloud

不论是商业应用还是用户应用，在业务初期都很简单，我们通常会把它实现为单体结构的应用。但是，随着业务逐渐发展，产品思想会变得越来越复杂，单体结构的应用也会越来越复杂。这就会给应用带来如下的几个问题：

- 代码结构混乱：业务复杂，导致代码量很大，管理会越来越困难。同时，这也会给业务的快速迭代带来巨大挑战；
- 开发效率变低：开发人员同时开发一套代码，很难避免代码冲突。开发过程会伴随着不断解决冲突的过程，这会严重的影响开发效率；
- 排查解决问题成本高：线上业务发现 bug，修复 bug 的过程可能很简单。但是，由于只有一套代码，需要重新编译、打包、上线，成本很高。

由于单体结构的应用随着系统复杂度的增高，会暴露出各种各样的问题。近些年来，微服务架构逐渐取代了单体架构，且这种趋势将会越来越流行。Spring Cloud是目前最常用的微服务开发框架，已经在企业级开发中大量的应用。

@$什么是Spring Cloud

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、智能路由、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并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装，屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

设计目标：**协调各个微服务，简化分布式系统开发。**

Spring Cloud优缺点

微服务的框架那么多比如：dubbo、Kubernetes，为什么就要使用Spring Cloud的呢?

优点：

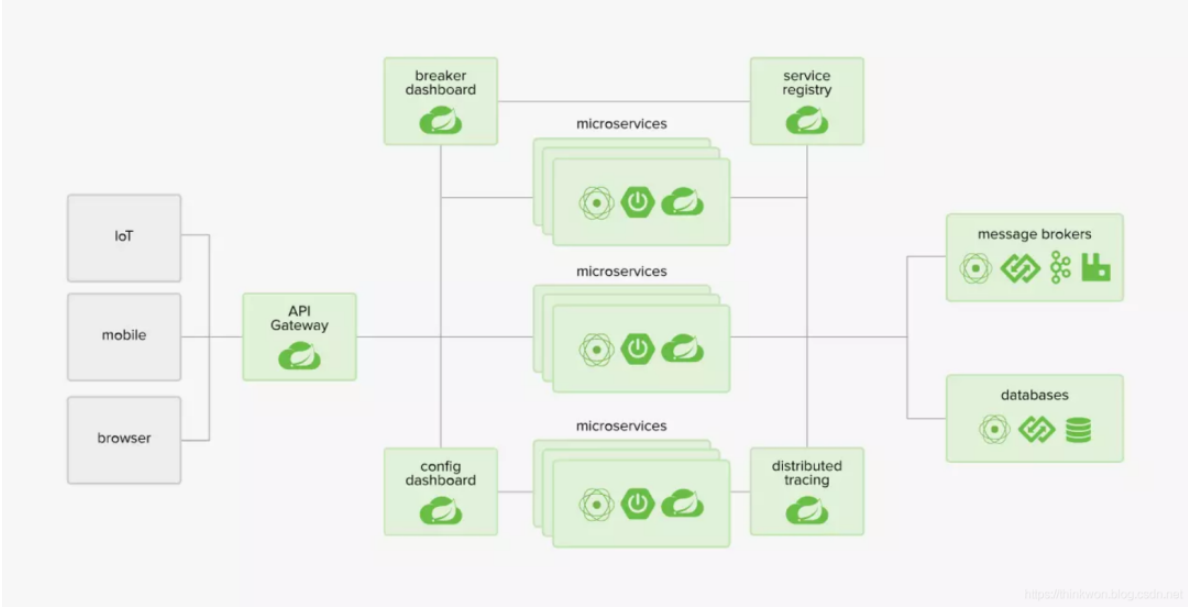
- 产出于Spring大家族，Spring在企业级开发框架中无人能敌，来头很大，可以保证后续的更新、完善
- 组件丰富，功能齐全。Spring Cloud 为微服务架构提供了非常完整的支持。例如、配置管理、服务发现、断路器、微服务网关等；
- Spring Cloud 社区活跃度很高，教程很丰富，遇到问题很容易找到解决方案
- 服务拆分粒度更细，耦合度比较低，有利于资源重复利用，有利于提高开发效率
- 可以更精准的制定优化服务方案，提高系统的可维护性
- 减轻团队的成本，可以并行开发，不用关注其他人怎么开发，先关注自己的开发
- 微服务可以是跨平台的，可以用任何一种语言开发
- 适于互联网时代，产品迭代周期更短

缺点：

- 微服务过多，治理成本高，不利于系统维护
- 分布式系统开发的成本高（容错，分布式事务等），对团队挑战大

总的来说优点大过于缺点，目前看来Spring Cloud是一套非常完善的分布式框架，目前很多企业开始用微服务、Spring Cloud的优势是显而易见的。因此对于想研究微服务架构的同学来说，学习Spring Cloud是一个不错的选择。

Spring Cloud整体架构



@$Spring Cloud主要项目有哪些

Spring Cloud的子项目，大致可分成两类，一类是对现有成熟框架"Spring Boot化"的封装和抽象，也是数量最多的项目；第二类是开发了一部分分布式系统的基础设施的实现，如Spring Cloud Stream扮演的就是kafka, ActiveMQ这样的角色。

Spring Cloud Config

集中配置管理工具，分布式系统中统一的外部配置管理，默认使用Git来存储配置，可以支持客户端配置的刷新及加密、解密操作。

Spring Cloud Netflix

Netflix OSS 开源组件集成，包括Eureka、Hystrix、Ribbon、Feign、Zuul等核心组件。

- Eureka：服务治理组件，包括服务端的注册中心和客户端的服务发现机制；
- Ribbon：负载均衡的服务调用组件，具有多种负载均衡调用策略；
- Hystrix：服务容错组件，实现了断路器模式，为依赖服务的出错和延迟提供了容错能力；
- Feign：基于Ribbon和Hystrix的声明式服务调用组件；
- Zuul：API网关组件，对请求提供路由及过滤功能。

Spring Cloud Bus

用于传播集群状态变化的消息总线，使用轻量级消息代理连接分布式系统中的节点，可以用来动态刷新集群中的服务配置。

Spring Cloud Consul

基于Hashicorp Consul的服务治理组件。

Spring Cloud Security

安全工具包，对Zuul代理中的负载均衡OAuth2客户端及登录认证进行支持。

Spring Cloud Gateway

新一代API网关组件，对请求提供路由及过滤功能。

Spring Cloud OpenFeign

基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC注解的接口实现用于服务调用，在Spring Cloud 2.0中已经取代Feign成为了一等公民。

SpringBoot和SpringCloud的区别?

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。SpringCloud将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

SpringBoot可以离开SpringCloud独立开发项目， 但是SpringCloud离不开SpringBoot ，属于依赖的关系。

@$Dubbo 和 Spring Cloud 有什么关系？Dubbo 和 Spring Cloud 有什么哪些区别?

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。

而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spring、Spring Boot 的优势，两个框架在目标就不一致，Dubbo 定位服务治理、Spring Cloud 是打造一个分布式的生态。

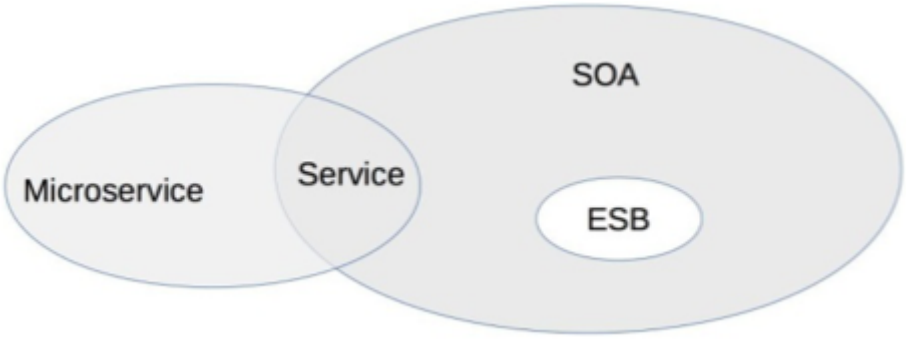
Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

Spring Cloud 是基于 Http 协议，使用 Restful 风格进行接口通信，相对来说 Http 请求会有更大的报文，占的带宽也会更多。但是 Restful 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体分析。

SOA和微服务的对比

微服务是一种和SOA相似但本质上不同的架构理念

- 相似点在于两者都关注“服务”，都是通过服务的拆分来解决可扩展性问题
- 本质上不同的地方在于几个核心理念的差异：是否有ESB（企业服务总线）、服务的粒度、架构设计的目标



SOA和微服务的对比

1、服务粒度

- SOA的服务粒度要粗一些，而微服务要细一些
- 例如，对一个大型企业来说，“员工管理系统”就是一个SOA架构中的服务；而如果采用微服务架构，则“员工管理系统”会被拆分为更多的服务，比如“员工信息管理”“员工考勤管理”“员工假期管理”和“员工福利管理”等更多服务

2、服务通信

- SOA采用了ESB作为服务间通信的关键组件，负责服务定义、服务路由、消息转换、消息传递，总体上是重量级的实现；微服务推荐使用统一的协议和格式，例如，RESTful协议、RPC协议，无须ESB这样的重量级实现

3、服务交付

- SOA对服务的交付并没有特殊要求，因为SOA更多考虑的是兼容已有的系统；微服务的架构理念要求“快速交付”，相应地要求采取自动化测试、持续集成、自动化部署等敏捷开发相关的最佳实践

4、应用场景

- SOA更加适合于庞大、负责异构的企业级系统，这类系统都发展多年，采用不同的企业级技术，有的是内部开发的，有的是外部购买的，无法完全推倒重来或者进行大规模的优化和重构，由于成本和影响太大，只能采用兼容的方式进行处理，而承担兼容任务的就是ESB
- 微服务更加适合于快速、轻量级、基于Web的互联网系统，这类系统业务变化快，需要快速尝试、快速交付；虽然开发技术可能差异很大（Java、C++、.NET等），但对外接口基本都是提供HTTP RESTful风格的接口，无须考虑在接口层进行类似SOA的ESB处理

SOA和微服务对比如下

对比维度	SOA	微服务
服务粒度	粗	细
服务通信	重量级，ESB	轻量级，例如，HTTP RESTful
服务交付	慢	快
应用场景	企业级	互联网

Nacos

什么是 Nacos

Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。

Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos 是构建以“服务”为中心的现代应用架构 (例如微服务范式、云原生范式) 的服务基础设施。

Nacos 的关键特性包括:

- 服务发现和服务健康监测

  Nacos 支持基于 DNS 和基于 RPC 的服务发现。

  Nacos 提供对服务的实时的健康检查，阻止向不健康的主机或服务实例发送请求。

- 动态配置服务

  动态配置服务可以让您以中心化、外部化和动态化的方式管理所有环境的应用配置和服务配置。

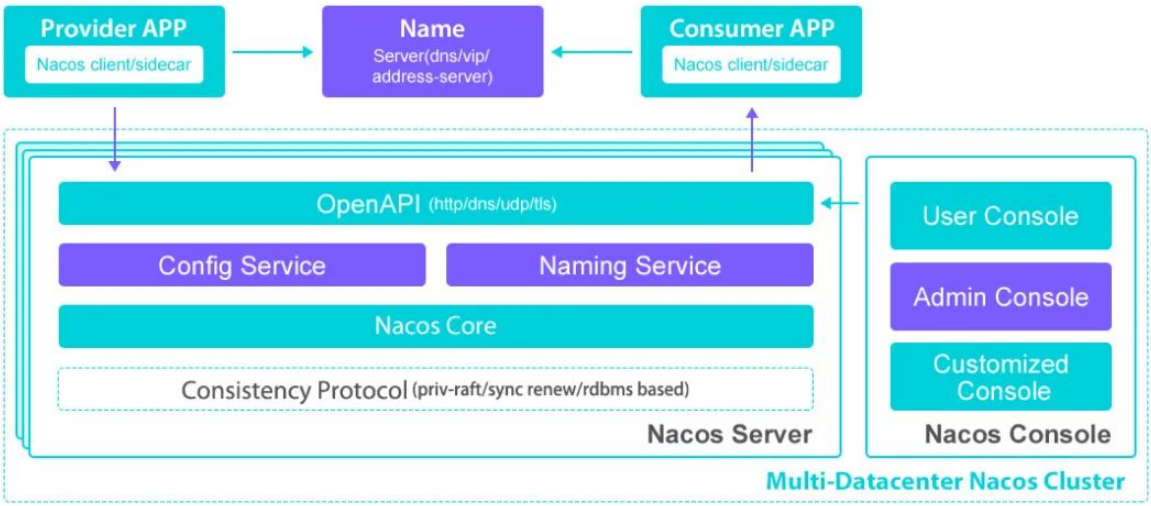
- 动态 DNS 服务

  动态 DNS 服务支持权重路由，让您更容易地实现中间层负载均衡、更灵活的路由策略、流量控制以及数据中心内网的简单DNS解析服务。

- 服务及其元数据管理

  Nacos 能让您从微服务平台建设的视角管理数据中心的所有服务及元数据，包括管理服务的描述、生命周期、服务的静态依赖分析、服务的健康状态、服务的流量管理、路由及安全策略、服务的 SLA 以及最首要的 metrics 统计数据。

Nacos基本架构及概念



服务 (Service)

服务是指一个或一组软件功能（例如特定信息的检索或一组操作的执行），其目的是不同的客户端可以为不同的目的重用（例如通过跨进程的网络调用）。Nacos 支持主流的服务生态，如 Kubernetes Service、gRPC|Dubbo RPC Service 或者 Spring Cloud RESTful Service.

服务注册中心 (Service Registry)

服务注册中心，它是服务，其实例及元数据的数据库。服务实例在启动时注册到服务注册表，并在关闭时注销。服务和路由器的客户端查询服务注册表以查找服务的可用实例。服务注册中心可能会调用服务实例的健康检查 API 来验证它是否能够处理请求。

服务元数据 (Service Metadata)

服务元数据是指包括服务端点(endpoints)、服务标签、服务版本号、服务实例权重、路由规则、安全策略等描述服务的数据

服务提供方 (Service Provider)

是指提供可复用和可调用服务的应用方

服务消费方 (Service Consumer)

是指会发起对某个服务调用的应用方

配置 (Configuration)

在系统开发过程中通常会将一些需要变更的参数、变量等从代码中分离出来独立管理，以独立的配置文件的形式存在。目的是让静态的系统工件或者交付物（如 WAR，JAR 包等）更好地和实际的物理运行环境进行适配。配置管理一般包含在系统部署的过程中，由系统管理员或者运维人员完成这个步骤。配置变更是调整系统运行时的行为的有效手段之一。

配置管理 (Configuration Management)

在数据中心中，系统中所有配置的编辑、存储、分发、变更管理、历史版本管理、变更审计等所有与配置相关的活动统称为配置管理。

名字服务 (Naming Service)

提供分布式系统中所有对象(Object)、实体(Entity)的“名字”到关联的元数据之间的映射管理服务，例如 ServiceName -> Endpoints Info, Distributed Lock Name -> Lock Owner/Status Info, DNS Domain Name -> IP List, 服务发现和 DNS 就是名字服务的2大场景。

配置服务 (Configuration Service)

在服务或者应用运行过程中，提供动态配置或者元数据以及配置管理的服务提供者。