2019
版

治愈系 Java 工程师面试指导课程

[Part5・SSM 和微服务框架]

[本教程梳理了 SSM 部分和 springboot、springcloud 的复习脉络，并且涵盖了框架部分的高频面试题；建议大家将目录结构打开，对照目录结构做一个复习的、提纲准备面试。]

1 Spring 简介

1.1 概念和组成

Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架，它是为了解决企业应用开发的复杂性而创建的。Spring 的核心是控制反转（IoC）和面向切面编程（AOP）。简单来说，Spring 是一个分层的 Java SE/EE full-stack(一站式) 轻量级开源框架。

Spring 由 20 多个模块组成，它们可以分为数据访问/集成（Data Access/Integration）、Web、面向切面编程（AOP, Aspects）、应用服务器设备管理（Instrumentation）、消息发送（Messaging）、核心容器（Core Container）和测试（Test）。

1.2 Spring 有四大特点

1.2.1 非侵入式

Spring 框架的 API 不会在业务逻辑上出现，即业务逻辑是 POJO

1.2.2 容器

Spring 作为一个容器，可以管理对象的生命周期、对象与对象之间的依赖关系。可以通过配置文件，来定义对象，以及设置与其他对象的依赖关系

1.2.3 IOC

控制反转（Inversion of Control），即调用者不创建被调用者的实例，而是由 Spring 容器创建被调用者，并注入调用者。 不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。
IOC: 控制反转也叫依赖注入。IOC 利用 java 反射机制，AOP 利用代理模式。IOC 概念看似很抽象，但是很容易理解。说简单点就是将对象交给容器管理，你只需要在 spring 配置文件中配置对应的 bean 以及设置相关的属性，让 spring 容器来生成类的实例对象以及管理对象。在 spring 容器启动的时候，spring 会把你在配置文件中配置的 bean 都初始化好，然后在你需要调用的时候，就把它已经初始化好的那些 bean 分配给你需要调用这些 bean 的类。

1.2.4 AOP

面向切面编程（AOP，Aspect Orient Programming），是一种编程思想，是面向对象编程 OOP 的补充。Spring 允许通过分离应用的业务逻辑与系统级服务（例如日志和事务管理）进行开发。
面向切面编程。（Aspect-Oriented Programming） 。AOP 可以说是对 OOP 的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。实现 AOP 的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码，属于静态代理。

治愈系Java工程师面试指导课程——samuel

1.3 Spring 的配置文件的加载

1.3.1 ApplicationContext 接口容器

A、配置文件在类路径下
若 Spring 配置文件存放在项目的类路径下，则使用 ClassPathXmlApplicationContext 实现类进行加载。

```
// 获取容器
ApplicationContext context =
        new ClassPathXmlApplicationContext("applicationContext.xml");
```

ApplicationContext
B、配置文件在本地目录中
若 Spring 配置文件存放在本地磁盘目录中，则使用 FileSystemXmlApplicationContext 实现类进行加载。

```
// 获取容器
ApplicationContext context =
        new FileSystemXmlApplicationContext("d:/applicationContext.xml");
```

C、配置文件在项目根路径下
若 Spring 配置文件存放在项目的根路径下，同样使用 FileSystemXmlApplicationContext 实现类进行加载。

```
// 获取容器
ApplicationContext context =
        new FileSystemXmlApplicationContext("applicationContext.xml");
```

1.3.2 BeanFactory 接口容器

BeanFactory 接口对象也可作为 Spring 容器出现。BeanFactory 接口是 ApplicationContext 接口的父类。若要创建 BeanFactory 容器，需要使用其实现类 XmlBeanFactory，而 Spring 配置文件以资源 Resouce 的形式出现在 XmlBeanFactory 类的构造器参数中。
Resouce 是一个接口，其具有两个实现类：
ClassPathResource：指定类路径下的资源文件
FileSystemResource：指定项目根路径或本地磁盘路径下的资源文件。

```
// 获取容器：获取类路径下的配置文件
BeanFactory factory =
        new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
/*
// 获取容器：获取当前项目根路径下的配置文件
BeanFactory factory =
        new XmlBeanFactory(new FileSystemResource("applicationContext.xml"));
```

尚硅谷Java工程师面试指导课程——samuel

1.3.3 两个接口容器的区别

虽然这两个接口容器所要加载的 Spring 配置文件是同一个文件，但在代码中的这两个容器对象却不是同一个对象，即不是同一个容器：它们对于容器内对象的装配（创建）时
机是不同的。

A、ApplicationContext 容器中对象的装配时机

ApplicationContext 容器，会在容器对象初始化时，将其中的所有对象一次性全部装配好。以后代码中若要使用到这些对象，只需从内存中直接获取即可。执行效率较高。但占用内存。

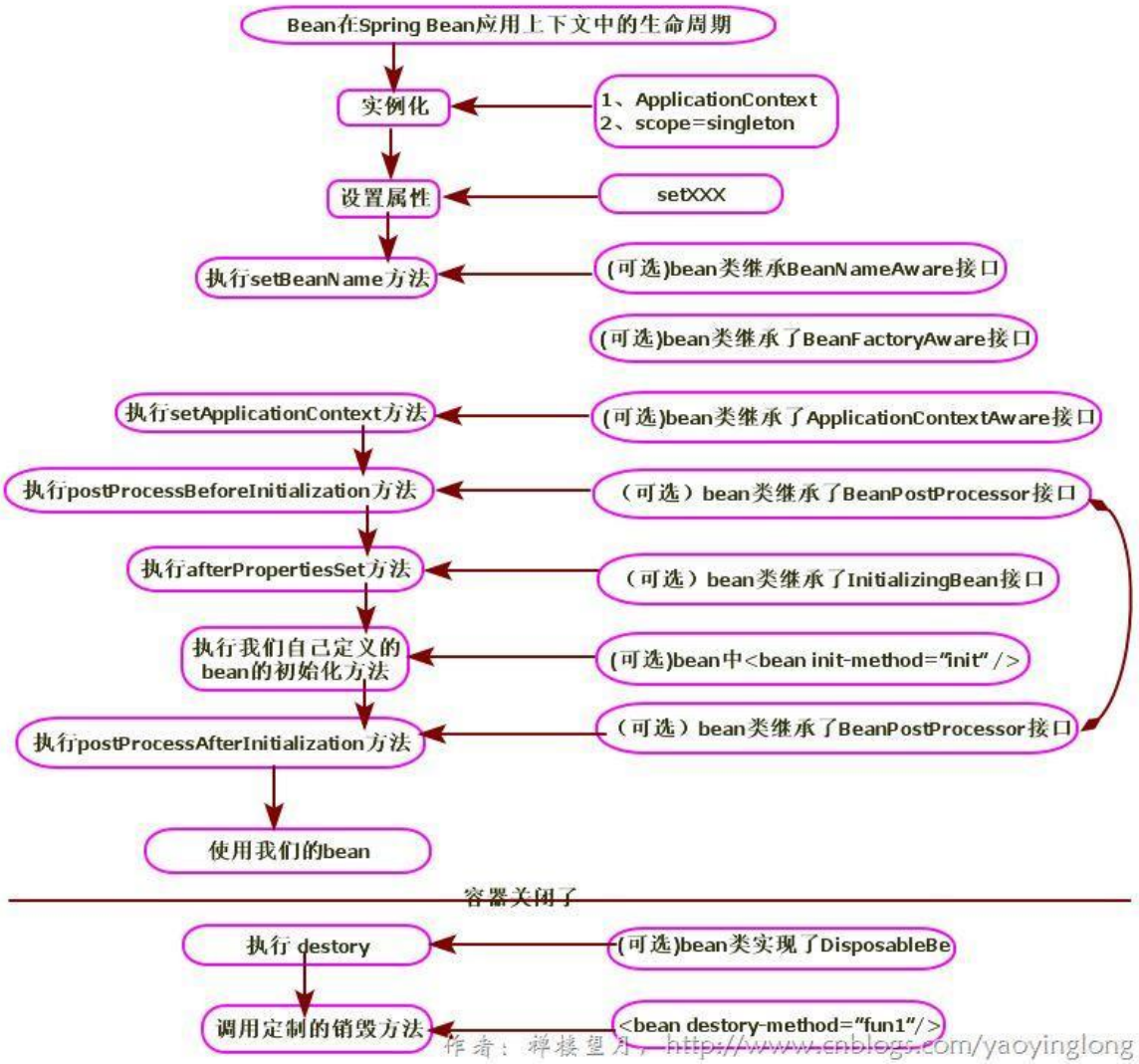
B、BeanFactory 容器中对象的装配时机

BeanFactory 容器，对容器中对象的装配与加载采用延迟加载策略，即在第一次调用 getBean()时，才真正装配该对象。

治愈系Java工程师面试指导课程——samuel

2 Spirng 作为容器

2.1 Spring Bean 生命周期



2.1.1 实例化

1. 实例化一个 Bean，也就是我们常说的 new。

2.1.2 IOC 依赖注入

2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入。

samuel

2.1.3 setBeanName 实现

3. 如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String) 方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值

2.1.4 BeanFactoryAware 实现

4. 如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory，setBeanFactory(BeanFactory)传递的是 Spring 工厂自身（可以用这个方式来获取其它 Bean， 只需在 Spring 配置文件中配置一个普通的 Bean 就可以）。

2.1.5 ApplicationContextAware 实现

5. 如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文（同样这个方式也 可以实现步骤 4 的内容，但比 4 更好，因为 ApplicationContext 是 BeanFactory 的子接 口，有更多的实现方法）

2.1.6 postProcessBeforeInitialization 接口实现-初始化预处理

6. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用 作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用那个的方法，也可以被应 用于内存或缓存技术。

2.1.7 init-method

7. 如果 Bean 在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法。

2.1.8 postProcessAfterInitialization

8. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用

2.1.9 postProcessAfterInitialization(Object obj, String s)方法。

注：以上工作完成以后就可以应用这个 Bean 了，那这个 Bean 是一个 Singleton 的，所以一 般情况下我们调用同一个 id 的 Bean 会是在内容地址相同的实例，当然在 Spring 配置文件中 也可以配置非 Singleton。

2.1.10 Destroy 过期自动清理阶段

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调 用那个其实现的 destroy()方法；

沿愈系Java工程师面试指导课程——samuel

2.1.11 destroy-method 自配置清理

10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的 销毁方法。

11. bean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制 初始化和注销方法。它们也有相应的注解（@PostConstruct 和@PreDestroy）。
<bean id="" class="" init-method="初始化方法" destroy-method="销毁方法">

2.1.12 Bean 的生命始末

在配置文件的<bean/>标签中增加如下属性：
init-method：指定初始化方法的方法名
destroy-method：指定销毁方法的方法名 destroy-method 的执行结果，需要满足两个条件：
（1）Bean 为 singleton，即单例
（2）要确保容器关闭。接口 ApplicationContext 没有 close()方法，但其实现类有。所以，可以将 ApplicationContext 强转为其实现类对象，或直接创建的就是实现类对象

2.2 容器中 Bean 的作用域

通过 scope 属性，为 Bean 指定特定的作用域。Spring 支持 5 种作用域。
（1）singleton：单态模式。即在整个 Spring 容器中，使用 singleton 定义的 Bean 将是单例的，只有一个实例。默认为单态的。
（2）prototype：原型模式。即每次使用 getBean 方法获取的同一个<bean />的实例都是一个新的实例。
（3）request：对于每次 HTTP 请求，都将会产生一个不同的 Bean 实例。
（4）session：对于每个不同的 HTTP session，都将产生一个不同的 Bean 实例。
（5）global session：每个全局的 HTTP session 对应一个 Bean 实例。典型情况下，仅在使用 portlet 集群时有效，多个 Web 应用共享一个 session。一般应用中，global-session 与 session 是等同的。
注意：
（1）对于 scope 的值 request、session 与 global session，只有在 Web 应用中使用 Spring 时，该作用域才有效。
（2）对于 scope 为 singleton 的单例模式，该 Bean 是在容器被创建时即被装配好了。
（3）对于 scope 为 prototype 的原型模式，Bean 实例是在代码中使用该 Bean 实例时才进行装配的

2.2.1 Spring 中的线程安全

Spring 是没有对 bean 的多线程安全问题作出任何保证和措施的。对于每个 bean 的线程安全问题，根本原因是每个 bean 自身的设计。不要在 bean 中声明任何有状态的实例变量或类变量。如果必须如此的话，那么就使用 ThreadLoad 把变量变为线程私有的，如果 bean 的实例变量或者类变量需要在多个线程之间共享，那么就只用使用 synchronized、lock、CAS 等这些实现线程同步的

治愈系Java工程师面试指导课程——samuel

2.3 Bean 后处理器

Bean 后处理器是一种特殊的 Bean，容器中所有的 Bean 在初始化时，均会自动执行该类的两个方法。由于该 Bean 是由其它 Bean 自动调用执行，不是程序员手工调用，故此 Bean 无须 id 属性。代码中需要自定义 Bean 后处理器类。该类就是实现了接口 BeanPostProcessor 的类。

2.4 <bean/>标签的 id 属性与 name 属性

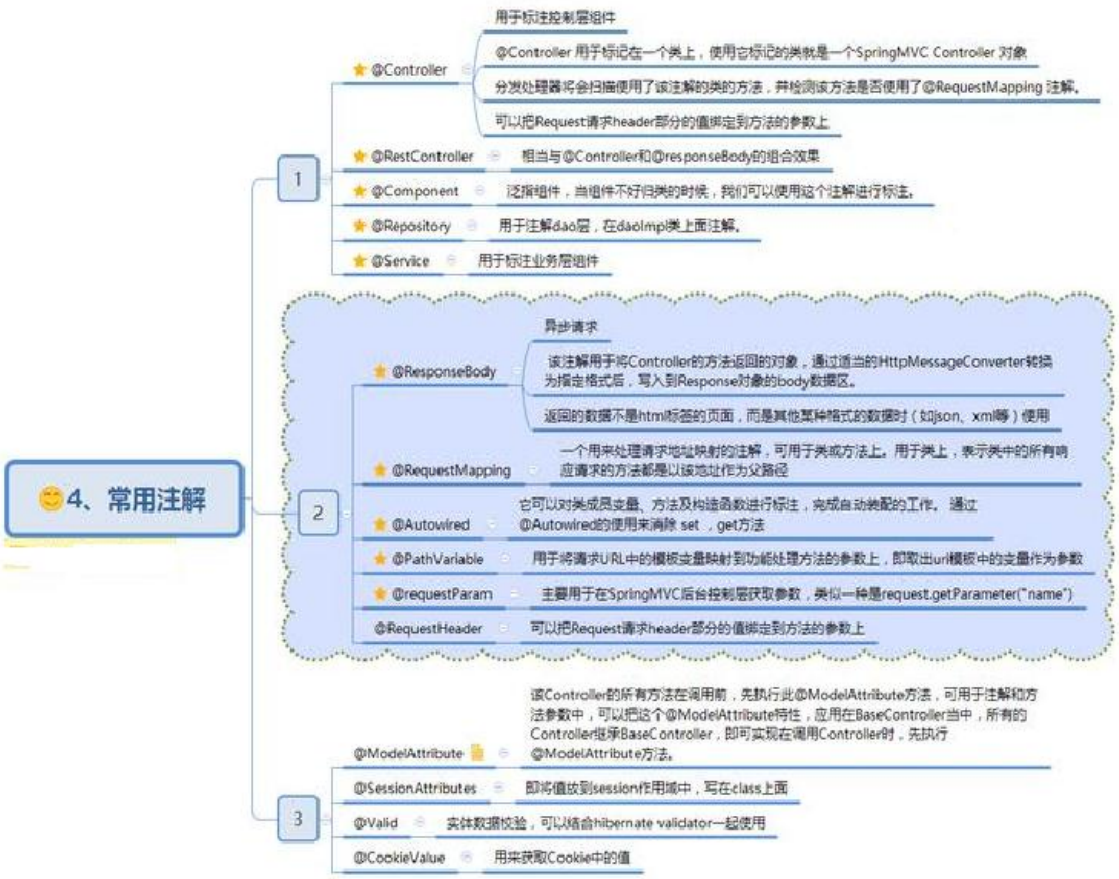
一般情况下，命名<bean/>使用 id 属性，而不使用 name 属性。在没有 id 属性的情况下，name 属性与 id 属性作用是相同的。但，当<bean/>中含有一些特殊字符时，就需要使用 name 属性了。

id 的命名需要满足 XML 对 ID 属性命名规范：必须以字母开头，可以包含字母、数字、下划线、连字符、句话、冒号。且要求名称在容器中必须唯一。

name 则可以包含各种字符，且对名称没有唯一性要求。若名称不唯一，则后面的会覆盖前面的。

2.5 Spring 常用注解

bean 注入与装配的的方式有很多种，可以通过 xml，get set 方式，构造函数或者注解等。简单易 用的方式就是使用 Spring 的注解了，Spring 提供了大量的注解方式。



@Component 和 @Bean 的区别是什么？

尚学堂课程--samuel

作用对象不同: @Component 注解作用于类，而@Bean 注解作用于方法。

@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 注解通常是我们在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，当我需要用它的时候还给我。

@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。

3 Spring 与 IOC

3.1 概念

IoC 是一个概念，是一种思想，其实现方式多种多样。当前比较流行的实现方式有两种：依赖注入和依赖查找。依赖注入方式应用更为广泛。

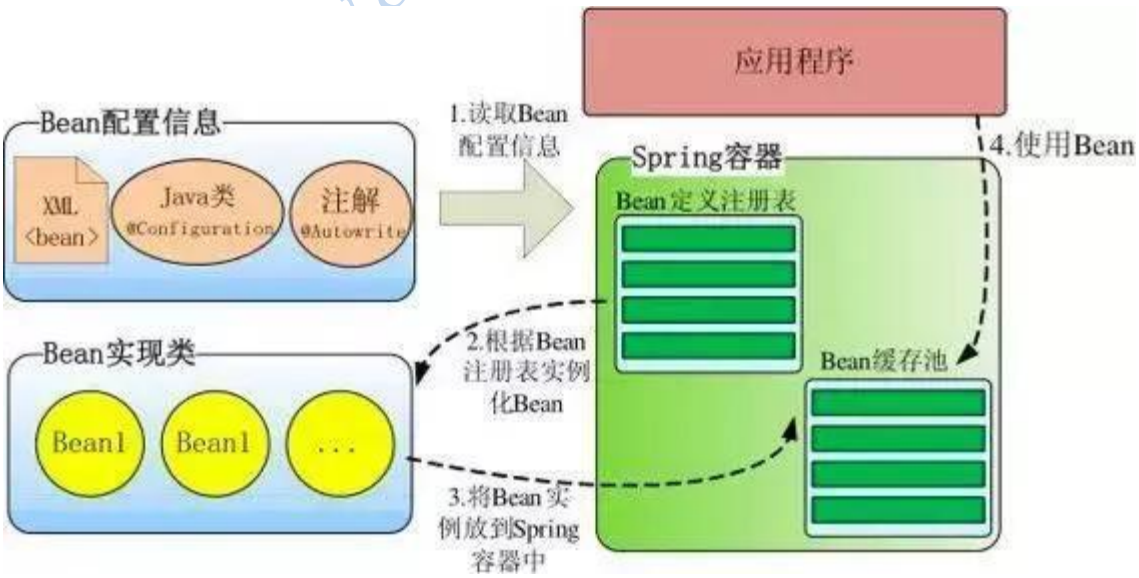
依赖查找：Dependency Lookup，DL，容器提供回调接口和上下文环境给组件，程序代码则需要提供具体的查找方式。比较典型的是依赖于 JNDI 系统的查找。

依赖注入：Dependency Injection，DI，目前最优秀的解耦方式，是指程序运行过程中，若需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部容器，由外部容器创建后传递给程序。Spring DI = 工厂 + 反射 + 配置文件

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。 Spring 的 IoC 容器在完成这些底层工作的基础上，还提供 了 Bean 实例缓存、生命周期管理、 Bean 实例代理、事件发布、资源装载等高级服务。

3.2 Spring IOC 过程

Spring 启动时读取应用程序提供的 Bean 配置信息，并在 Spring 容器中生成一份相应的 Bean 配 置注册表，然后根据这张注册表实例化 Bean，装配好 Bean 之间的依赖关系，为上层应用提供准 备就绪的运行环境。其中 Bean 缓存池为 HashMap 实现



Spring IOC 的初始化过程

3.3 Spring 依赖注入四种方式

3.3.1 构造器注入

```
/*带参数，方便利用构造器进行注入*/
public CatDaoImpl(String message){
this. message = message;
}
<bean id="CatDaoImpl" class="com.CatDaoImpl">
<constructor-arg value=" message "></constructor-arg>  </bean>
setter 方法注入
public class Id {
private int id;
public int getId() {  return id;  }
public void setId(int id) {  this.id = id;  }  }
<bean id="id" class="com.id ">  <property name="id" value="123"></property> </bean>
```

3.3.2 静态工厂注入

静态工厂顾名思义，就是通过调用静态工厂的方法来获取自己需要的对象，为了让 spring 管理所 有对象，我们不能直接通过"工程类.静态方法()"来获取对象，而是依然通过 spring 注入的形式获 取：

```
public class DaoFactory {  //静态工厂
public static final FactoryDao getStaticFactoryDaoImpl(){
return new StaticFacotryDaoImpl();
}
}
public class SpringAction {
private FactoryDao staticFactoryDao;  //注入对象
//注入对象的  set 方法
public void setStaticFactoryDao(FactoryDao staticFactoryDao) {
this.staticFactoryDao = staticFactoryDao;
}
}
//factory-method="getStaticFactoryDaoImpl"指定调用哪个工厂方法
<bean name="springAction" class=" SpringAction" >
<!--使用静态工厂的方法注入对象,对应下面的配置文件-->
<property name="staticFactoryDao" ref="staticFactoryDao"></property>  </bean>
<!--此处获取对象的方式是从工厂类中获取静态方法-->
```

沧海系Java工程师面试指导课程——samuel

<bean name="staticFactoryDao" class="DaoFactory"
factory-method="getStaticFactoryDaoImpl"></bean>

3.3.3 实例工厂

实例工厂的意思是获取对象实例的方法不是静态的，所以你需要首先 new 工厂类，再调用普通的 实例方法：

```
public class DaoFactory {    //实例工厂
public FactoryDao getFactoryDaoImpl(){   return new FactoryDaoImpl();   }
}
public class SpringAction {
private FactoryDao factoryDao;        //注入对象
public void setFactoryDao(FactoryDao factoryDao) {
this.factoryDao = factoryDao;
}
}
```

<bean name="springAction" class="SpringAction">
<!--使用实例工厂的方法注入对象,对应下面的配置文件-->
<property name="factoryDao" ref="factoryDao"></property> </bean>
<!--此处获取对象的方式是从工厂类中获取实例方法-->
<bean name="daoFactory" class="com.DaoFactory"></bean> <bean name="factoryDao" factory-bean="daoFactory"
factory-method="getFactoryDaoImpl"></bean>

3.4 五种不同方式的自动装配

Spring 装配包括手动装配和自动装配，手动装配是有基于 xml 装配、构造方法、setter 方法等
自动装配有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

1. no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。
2. byName：通过参数名 自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设 置成 byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
3. byType：通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被 设置成 byType，之后容器试图匹配、装配和该 bean 的属性具有相同类型的 bean。如果有多 个 bean 符合条件，则抛出错误。
4. constructor：这个方式类似于 byType， 但是要提供给构造器参数，如果没有确定的带参数 的构造器参数类型，将会抛出异常。
5. autodetect：首先尝试使用 constructor 来自动装配，如果无法工作，则使用 byType 方式。

3.5 基于注解的 DI

```
@Component
    @Repository
    @Service
   @Controller
@Scope
```

冷怠为Java工程师面试指导课程——samuel

@Value
@Autowired
@Autowired(required=false)与@Qualifier
@Resource 根据 name 属性有无
@PostConstruct 与@PreDestroy
@Configuration

4 Spring 与 AOP

4.1 AOP 概述

4.1.1 AOP 简介

AOP（Aspect Orient Programming），面向切面编程，是面向对象编程 OOP 的一种补充。 面向对象编程是从静态角度考虑程序的结构，而面向切面编程是从动态角度考虑程序运行过程。

AOP 底层，就是采用动态代理模式实现的。采用了两种代理：JDK 的动态代理，与 CGLIB 的动态代理。

面向切面编程，就是将交叉业务逻辑封装成切面，利用 AOP 容器的功能将切面织入到主业务逻辑中。所谓交叉业务逻辑是指，通用的、与主业务逻辑无关的代码，如安全检查、事务日志等。

"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块， 并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共 同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未 来的可操作性和可维护性。

使用"横切"技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流 程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生 在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP 的作用在于分离系统 中的各种关注点，将核心关注点和横切关注点分离开来。

治愈系Java工程师面试宝典课程——samuel

4.1.2 AOP 编程术语

4.1.2.1 切面（Aspect）

切面泛指交叉业务逻辑。上例中的事务处理、日志处理就可以理解为切面。常用的切面有通知与顾问。实际就是对主业务逻辑的一种增强。

4.1.2.2 目标对象（Target）

目标对象指将要被增强的对象。即包含主业务逻辑的类的对象。上例中的 StudentServiceImpl 的对象若被增强，则该类称为目标类，该类对象称为目标对象。当然，不被增强，也就无所谓目标不目标了。

4.1.2.3 织入（Weaving）

织入是指将切面代码插入到目标对象的过程。上例中 MyInvocationHandler 类中的 invoke()方法完成的工作，就可以称为织入。

4.1.2.4 引入（introduction）

在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法 或字段。

4.1.2.5 连接点（JoinPoint）

连接点指切面可以织入的位置。上例中 IService 中的 doSome()与 doOther()均为连接点。就是业务逻辑的方法

4.1.2.6 切入点（Pointcut）

切入点首先是连接点，给连接点织入了功能增强就是切入点。

切入点指切面具体织入的位置。在 StudentServiceImpl 类中，若 doSome()将被增强，而 doOther()不被增强，则 doSome()为切入点，而 doOther()仅为连接点。被标记为 final 的方法是不能作为连接点与切入点的。因为最终的是不能被修改的，不能被增强的。

4.1.2.7 通知（Advice）

通知是切面的一种实现，可以完成简单织入功能（织入功能就是在这里完成的）。上例中的 MyInvocationHandler 就可以理解为是一种通知。换个角度来说，通知定义了增强代码切入到目标代码的时间点，是目标方法执行之前执行，还是之后执行等。通知类型不同，切入时间不同。

切入点定义切入的位置，通知定义切入的时间。

4.1.2.8 顾问（Advisor）

顾问是切面的另一种实现，能够将通知以更为复杂的方式织入到目标对象中，是将通知包装为更复杂切面的装配器。

4.1.2.9 AOP 代理（Proxy）

代理对象 = 目标对象 + 交叉业务逻辑

治愈系Java工程师面试指导课程——samuel

4.1.3 AOP 主要应用场景

1. Authentication 权限
2. Caching 缓存
3. Context passing 内容传递
4. Error handling 错误处理
5. Lazy loading 懒加载
6. Debugging 调试
7. logging, tracing, profiling and monitoring 记录跟踪 优化 校准
8. Performance optimization 性能优化
9. Persistence 持久化
10. Resource pooling 资源池
11. Synchronization 同步
12. Transactions 事务

4.2 AOP 两种代理方式

代理模式

是常用的 java 设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。 动态代理类的字节码在程序运行时由 Java 反射机制动态生成，无需程序员手工编写它的源代码。

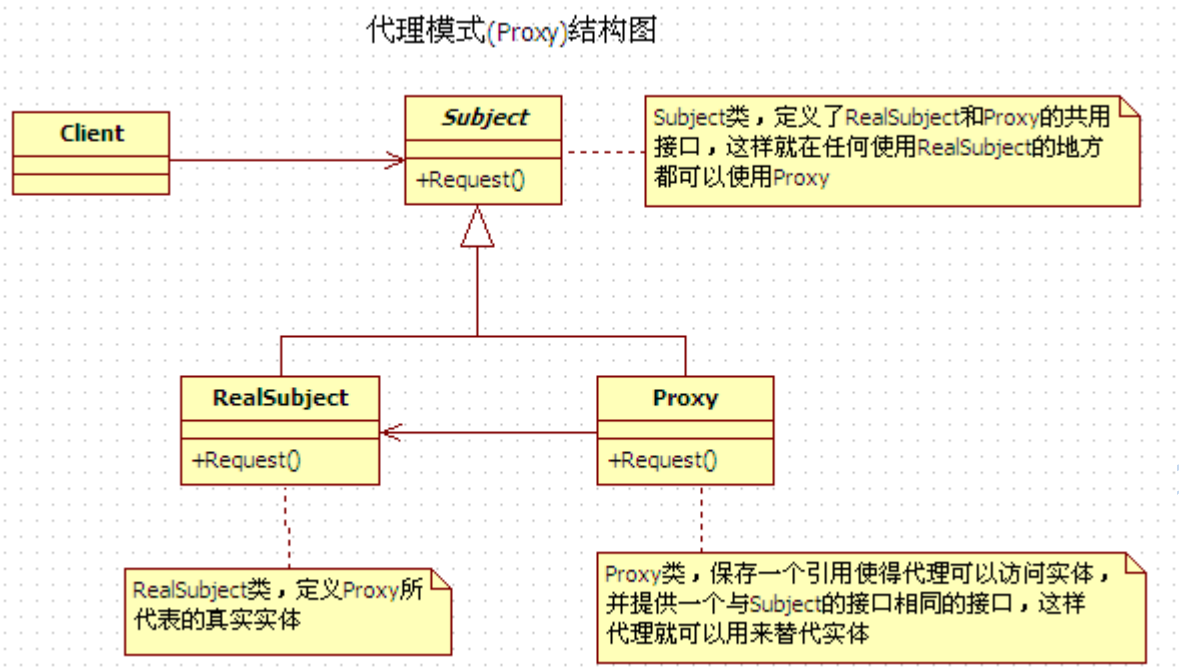
动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为 Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的 Proxy 类和 InvocationHandler 接口提供了生成动态代理类的能力。 JDK 的动态代理依靠接口实现，如果有些类并没有实现接口，则不能使用 JDK 代理，这就要使用 cglib 动态代理了

Cglib 动态代理

JDK 的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现 JDK 的动态代理，cglib 是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对 final 修饰的类进行代理

代理模式：代理类和被代理类实现共同的接口（或继承），代理类中存有指向被代理类的索引，实际执行时通过调用代理类的方法、实际执行的是被代理类的方法。

治愈系Java工程师的终极导课程——samuel



而 AOP，是通过动态代理实现的。Spring 提供了两种方式来生成代理对象：JDKProxy 和 Cglib，具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口， 则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。

4.2.1 区别

JDK 动态代理只能对实现了接口的类生成代理，而不能针对类.

JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。 InvocationHandler 是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类 的代码，动态将横切逻辑和业务逻辑编制在一起。Proxy 利用 InvocationHandler 动态创建 一个符合某一接口的实例，生成目标类的代理对象。

CGLIB 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法（继承）

CGLib 全称为 Code Generation Library，是一个强大的高性能，高质量的代码生成类库， 可以在运行期扩展 Java 类与实现 Java 接口，CGLib 封装了 asm，可以再运行期动态生成新 的 class。和 JDK 动态代理相比较：JDK 创建代理有一个限制，就是只能为接口创建代理实例， 而对于没有通过接口定义业务方法的类，则可以通过 CGLib 创建动态代理。

4.2.2 Spring 在选择用 JDK 还是 CGLiB 的依据：

(1)当 Bean 实现接口时，Spring 就会用 JDK 的动态代理

(2)当 Bean 没有实现接口时，Spring 使用 CGlib 是实现

(3)可以强制使用 CGlib（在 spring 配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）

4.2.3 CGlib 比 JDK 快？

(1)使用 CGLib 实现动态代理，CGLib 底层采用 ASM 字节码生成框架，使用字节码技术生成代理类，比使用 Java 反射效率要高。唯一需要注意的是，CGLib 不能对声明为 final 的方法进行代理，因为 CGLib 原理是动态生成被代理类的子类。

(2)在对 JDK 动态代理与 CGlib 动态代理的代码实验中看，1W 次执行下，JDK7 及 8 的动态代理性能比 CGlib 要好 20%左右。

4.3 通知 Advice

ProxyFactoryBean 不是代理类，而是代理对象生成器。对于测试类，需要注意，从容器中获取的是代理对象，而非目标对象。

4.3.1 前置通知 MethodBeforeAdvice

定义前置通知，需要实现 MethodBeforeAdvice 接口。该接口中有一个方法 before()，会在目标方法执行之前执行。前置通知的特点：

在目标方法执行之前先执行。

不改变目标方法的执行流程，前置通知代码不能阻止目标方法执行。

不改变目标方法执行的结果。

4.3.2 后置通知 AfterReturningAdvice

定义后置通知，需要实现接口 AfterReturningAdvice。该接口中有一个方法 afterReturning()会在目标方法执行之后执行。后置通知的特点：

在目标方法执行之后执行。

不改变目标方法的执行流程，后置通知代码不能阻止目标方法执行。

不改变目标方法执行的结果。

4.3.3 环绕通知 MethodInterceptor

定义环绕通知，需要实现 MethodInterceptor 接口。环绕通知，也叫方法拦截器，可以在目标方法调用之前及之后做处理，可以改变目标方法的返回值，也可以改变程序执行流程。

沿途学Java工程师面试指导课程——samuel

4.3.4 异常通知 ThrowsAdvice

定义异常通知，需要实现 ThrowsAdvice 接口。该接口的主要作用是，在目标方法抛出异常后，根据异常的不同做出相应的处理。当该接口处理完异常后，会简单地将异常再次抛出给目标方法。

不过，这个接口较为特殊，从形式上看，该接口中没有必须要实现的方法。但，这个接口却确实有必须要实现的方法 afterThrowing()。

4.3.5 给目标方法织入多个切面

4.3.6 无接口的 CGLIB 代理生成

直接不用加<property name=”interfaces” value=”…..”>就可以了

4.3.7 有接口的 CGLIB 代理生成

增加<property name=” proxyTargetClass” value=”true”>
或者<property name=” optimize” value=”true”>

4.4 顾问 Advisor

通知（Advice）是 Spring 提供的一种切面（Aspect）。但其功能过于简单：只能将切面织入到目标类的所有目标方法中，无法完成将切面织入到指定目标方法中。

顾问（Advisor）是 Spring 提供的另一种切面。其可以完成更为复杂的切面织入功能。PointcutAdvisor 是顾问的一种，可以指定具体的切入点。顾问将通知进行了包装，会根据不同的通知类型，在不同的时间点，将切面织入到不同的切入点。

PointcutAdvisor 接口有两个较为常用的实现类：

NameMatchMethodPointcutAdvisor 名称匹配方法切入点顾问

RegexpMethodPointcutAdvisor 正则表达式匹配方法切入点顾问

4.4.1 NameMatchMethodPointcutAdvisor

NameMatchMethodPointcutAdvisor，即名称匹配方法切入点顾问。容器可根据配置文件中指定的方法名来设置切入点。

代码不用修改，只在配置文件中注册一个顾问，然后使用通知属性 advice 与切入点的方法名 mappedName 对其进行配置。代理中的切面，使用这个顾问即可。

4.4.2 RegexpMethodPointcutAdvisor

RegexpMethodPointcutAdvisor，即正则表达式方法顾问。容器可根据正则表达式来设置切入点。注意，与正则表达式进行匹配的对象是接口中的方法名，而非目标类（接口的实

传智播客Java工程师面试指导课程——samuel

现类）的方法名。

注意：pattern 可以写成 patterns

4.4.3 自动代理生成器

4.4.3.1 默认 advisor 自动代理生成器

4.4.3.2 Bean 名称自动代理生成器

4.5 AspectJ 对 AOP 的实现

AOP 是运行时增强是基于代理实现的，AspectJ 是编译时增强是基于字节吧操作。AOP 已经集成了 AspectJ，AspectJ 算是 java 体系中最完成的 AOP 框架了

AspectJ 是一个面向切面的框架，它扩展了 Java 语言。AspectJ 定义了 AOP 语法，它有一个专门的编译器用来生成遵守 Java 字节编码规范的 Class 文件。

AspectJ 除了提供了六种通知外，还定义了专门的表达式用于指定切入点。表达式的原型是

```
execution ( [modifiers-pattern]   访问权限类型
            ret-type-pattern   返回值类型
            [declaring-type-pattern]   全限定性类名
            name-pattern(param-pattern)  方法名(参数名)
            [throws-pattern]   抛出异常类型
          )
```

符号	意义
*	0至多个任意字符
..	用在方法参数中，表示任意多个参数 用在包名后，表示当前包及其子包路径
+	用在类名后，表示当前类及其子类 用在接口后，表示当前接口及其实现类

治愈系Java工程师养成计划课程——samuel

4.5.1 AspectJ 基于注解的 AOP 实现

4.5.1.1 @Before

4.5.1.2 @AfterReturning

4.5.1.3 @Around

4.5.1.4 @AfterThrowing

4.5.1.5 @After

4.5.1.6 @Pointcut

4.5.2 AspectJ 基于 XML 的 AOP 实现

5 Spring 的 IOC 和 AOP 的应用

5.1 Spring 中体现的设计模式

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。
模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
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。
……

咨询宗Java工程师面试指导课程——samuel

5.2 Spring 与 JDBC 模板(IOC)

5.2.1 数据源的配置

5.2.1.1 Spring 默认 DriverManagerDataSource

5.2.1.2 DBCP 数据源 BasicDataSource

5.2.1.3 C3P0 数据源 ComboPooledDataSource

5.2.2 从属性文件读取数据库连接信息

5.2.2.1 <bean/>方式-使用 class 为 PropertyPlaceholderConfigurer

5.2.2.2 <context:property-placeholder/>方式

5.2.3 配置 JDBC 模板-使用 class 为 JdbcTemplate

5.2.4 Dao 实现类继承 JdbcDaoSupport 类

在 Spring 配置文件中，对于 JDBC 模板对象的配置完全可以省去，而是在 Dao 实现类中直接注入数据源对象。这样会让系统自动创建 JDBC 模板对象。

JdbcTemplate 对象是多例的，即系统会为每一个使用模板对象的线程（方法）创建一个 JdbcTemplate 实例，并且在该线程（方法）结束时，自动释放 JdbcTemplate 实例。所以在每次使用 JdbcTemplate 对象时，都需要通过 getJdbcTemplate() 方法获取。

5.3 Spring 的事务管理(AOP)

5.3.1 Spring 事务管理 API

5.3.1.1 事务管理器接口

事务管理器是 PlatformTransactionManager 接口对象。其主要用于完成事务的提交、回滚，及获取事务的状态信息…。常用的两个实现类：

治愈系Java工程师面试指导课程——samuel

DataSourceTransactionManager ：使用 JDBC 或 iBatis 进行持久化数据时使用。

HibernateTransactionManager ：使用 Hibernate 进行持久化数据时使用。

Spring 的回滚方式是： 发生运行时异常时回滚，发生受查异常 时提交。不过，对于受查异常，程序员也可以 手工 设置其回滚方式

5.3.1.2 五个事务隔离级别常量

这些常量均是以 ISOLATION_开头。即形如 ISOLATION_XXX。

DEFAULT：采用 DB 默认的事务隔离级别。MySql 的默认为 REPEATABLE_READ；Oracle 默认为 READ_COMMITTED。

READ_UNCOMMITTED：读未提交。未解决任何并发问题。

READ_COMMITTED：读已提交。解决脏读，存在不可重复读与幻读。

REPEATABLE_READ：可重复读。解决脏读、不可重复读，存在幻读

SERIALIZABLE：串行化。不存在并发问题。

5.3.1.3 七个事务传播行为常量

所谓事务传播行为是指，处于不同事务中的方法在相互调用时，执行期间事务的维护情况。如，A 事务中的方法 doSome()调用 B 事务中的方法 doOther()，在调用执行期间事务的维护情况，就称为事务传播行为。事务传播行为是加在方法上的。

事务传播行为常量都是以 PROPAGATION_ 开头，形如 PROPAGATION_XXX。

REQUIRED：指定的方法必须在事务内执行。若当前存在事务，就加入到当前事务中；若当前没有事务，则创建一个新事务。这种传播行为是最常见的选择，也是 Spring 默认的事务传播行为。

如该传播行为加在 doOther()方法上。若 doSome()方法在执行时就是在事务内的，则 doOther()方法的执行也加入到该事务内执行。若 doSome()方法没有在事务内执行，则 doOther()方法会创建一个事务，并在其中执行。

SUPPORTS：指定的方法支持当前事务，但若当前没有事务，也可以以非事务方式执行。

MANDATORY：指定的方法必须在当前事务内执行，若当前没有事务，则直接抛出异常。

REQUIRES_NEW：总是新建一个事务，若当前存在事务，就将当前事务挂起，直到新事务执行完毕。

NOT_SUPPORTED：指定的方法不能在事务环境中执行，若当前存在事务，就将当前事务挂起。

NEVER：指定的方法不能在事务环境下执行，若当前存在事务，就直接抛出异常。

NESTED：指定的方法必须在事务内执行。若当前存在事务，则在嵌套事务内执行；若当前没有事务，则创建一个新事务。

5.3.1.4 默认事务超时时限

常量 TIMEOUT_DEFAULT 定义了事务底层默认的超时时限，及不支持事务超时时限设置的 none 值。

沟壑系Java工程师面试指导课程——samuel

注意，事务的超时时限起作用的条件比较多，且超时的时间计算点较复杂。所以，该值一般就使用默认值即可。

5.3.2 使用 Spring 的事务代理工厂管理事务

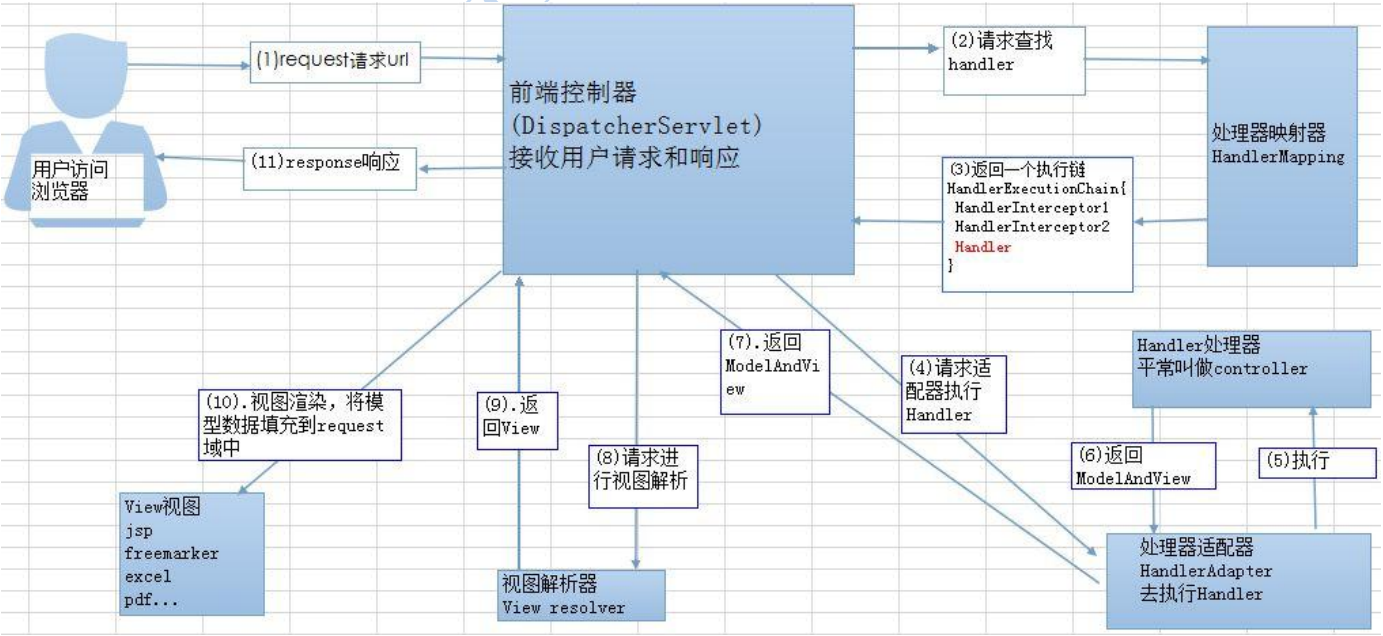
5.3.3 使用 Spring 的事务注解管理事务

5.3.4 使用 AspectJ 的 AOP 配置管理事务

程师面试指导课程——samuel

6 Spring MVC

6.1 原理



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. DispaterServlet 把返回的 Model 传给 View（视图渲染）。
8. 把 View 返回给请求者（浏览器）

Spring 的模型-视图-控制器（MVC）框架是围绕一个 DispatcherServlet 来设计的，这个 Servlet 会把请求分发给各个处理器，并支持可配置的处理器映射、视图渲染、本地化、时区与主题渲染 等，甚至还能支持文件上传。

6.2 常用注解

6.3 SpringMVC 的优点

是基于组件技术的.全部的应用对象,无论控制器和视图,还是业务对象之类的都是 java 组件.并且和 Spring 提供的其他基础结构紧密集成
与 Spring 框架天生整合,无框架兼容问题 与 Struts2 相比安全性高 配置量小、开发效率高

6.4 SpringMVC 与 Struts2 的区别

springmvc 的入口是一个 servlet 即前端控制器，而 struts2 入口是一个 filter 过虑器。
springmvc 是基于方法开发，传递参数是通过方法形参，可以设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。
Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据。 springmvc 通过参数解析器是将 request 对象内容进行解析成方法形参，将响应数据和页面封装成 ModelAndView 对象，最后又将模型数据通过 request 对象传输到页面。 Jsp 视图解析器默认使用 jstl。

7 Mybatis

7.1 什么是 Mybatis？

（1）Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。

（2）MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

（3）通过 xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。（从执行 sql 到返回 result 的过程）。

7.2 Mybaits 的优点

（1）基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。
（2）与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；
（3）很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）。
（4）能够与 Spring 很好的集成；

沿途教Java工程师面试指导课程-samuel

（5）提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

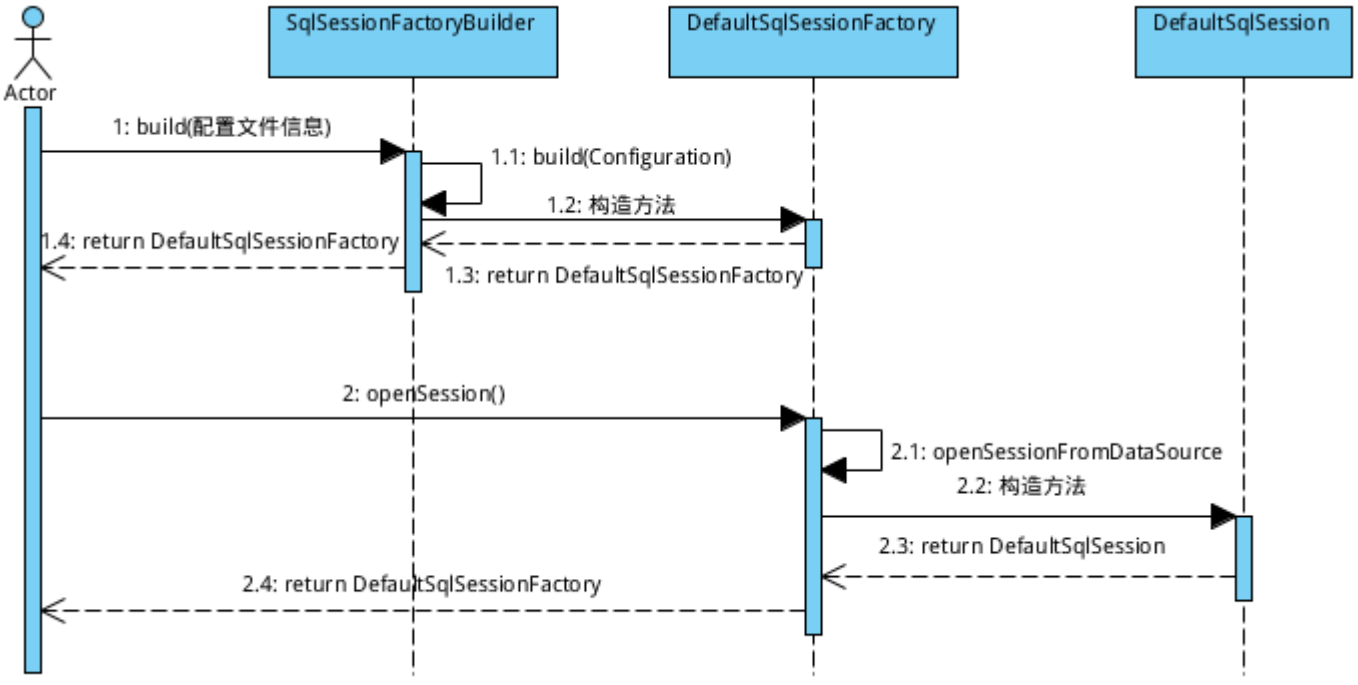
7.3 MyBatis 框架的缺点

（1）SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求。
（2）SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

7.4 MyBatis 框架适用场合

（1）MyBatis 专注于 SQL 本身，是一个足够灵活的 DAO 层解决方案。
（2）对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis 将是不错的选择

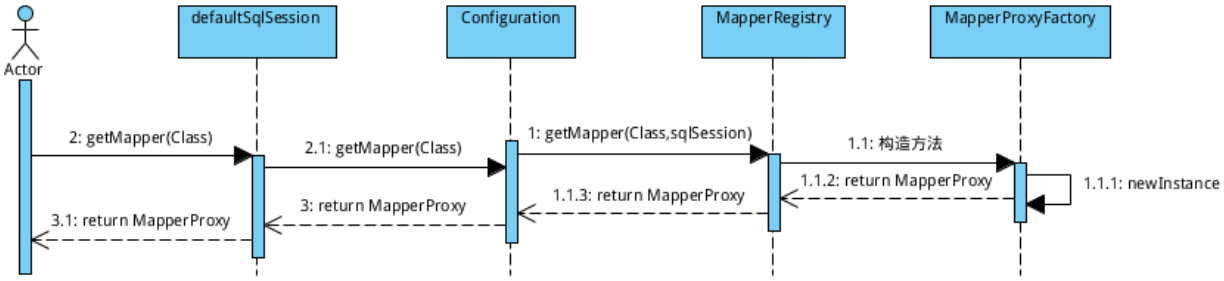
7.5 Mybatis 的执行过程



（1）首先，SqlSessionFactoryBuilder 去读取 mybatis 的配置文件，然后 build 一个 DefaultSqlSessionFactory。
（2）当我们获取到 SqlSessionFactory 之后，就可以通过 SqlSessionFactory 去获取 SqlSession 对象
（3）拿到 SqlSession 对象以后就可以调用 SqlSession 中一系列的 select..., insert..., update..., delete...方法轻松自如的进行 CRUD 操作了。 就这样？ 那咱配置的映射文件去哪儿了？ 别急， 咱们接着往下看：

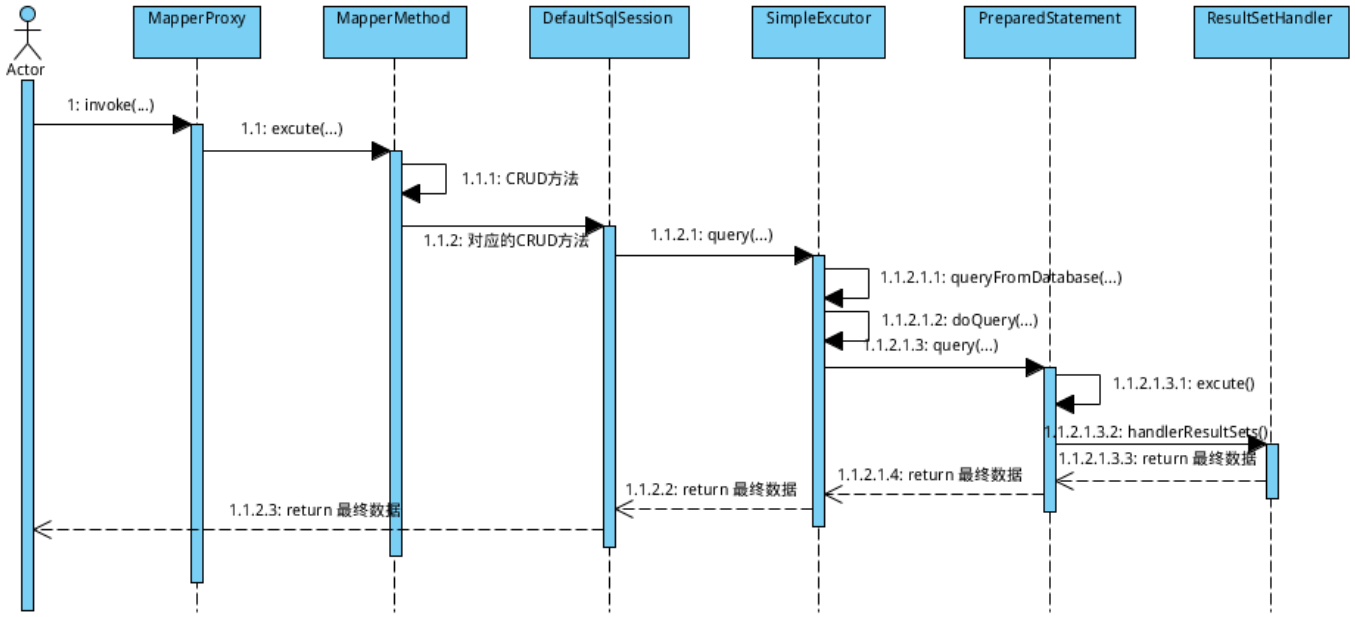
指导课程——samuel

7.5.1 利器之 MapperProxy



在 mybatis 中，通过 MapperProxy 动态代理咱们的 dao， 也就是说， 当咱们执行自己写的 dao 里面的方法的时候，其实是对应的 mapperProxy 在代理。那么，咱们就看看怎么获取 MapperProxy 对象吧：MapperProxyFactory 的动态代理，咱们就可以方便地使用 dao 接口

7.5.2 Excutor



咱们拿到了 MapperProxy，每个 MapperProxy 对应一个 dao 接口， 那么咱们在使用的时候，MapperProxy 是怎么做的呢？

7.6 MyBatis 与 Hibernate 有哪些不同？

（1）Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句。

（2）Mybatis 直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套 sql 映射文件，工作量大。

（3）Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用 hibernate 开发可以节

省很多代码，提高效率。
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。

7.7 #{}和${}的区别是什么？

#{}是预编译处理，${}是字符串替换。
Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；
Mybatis 在处理${}时，就是把${}替换成变量的值。
使用#{}可以有效的防止 SQL 注入，提高系统安全性。

7.8 当实体类中的属性名和表中的字段名不一样 ，怎么办 ？

第 1 种： 通过在查询的 sql 语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。
第 2 种： 通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

7.9 关于动态 mapper 的理解

（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？）

Dao 接口即 Mapper 接口。接口的全限名，就是映射文件中的 namespace 的值；接口的方法名，就是映射文件中 Mapper 的 Statement 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，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MapperStatement。在 Mybatis 中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个 MapperStatement 对象。

Mapper 接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。Mapper 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 Mapper 接口生成代理对象 proxy，代理对象会拦截接口方法，转而执行 MapperStatement 所代表的 sql，然后将 sql 执行结果返回。

7.9.1 使用 MyBatis 的 mapper 接口调用时有哪些要求？

尚硅谷Java工程师面试指导课程 - samuel

① Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同；
② Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同；
③ Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同；
④ Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

7.10 如何获取自动生成的(主)键值?

insert 方法总是返回一个 int 值 ，这个值代表的是插入的行数。

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。

示例：

```
<insert id=”insertname” usegeneratedkeys=”true” keyproperty=”id”>
    insert into names (name) values (#{name})
</insert>
```

7.11 Mybatis 动态 sql 有什么用？执行原理？有哪些动态 sql?

Mybatis 动态 sql 可以在 Xml 映射文件内，以标签的形式编写动态 sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接 sql 的功能。

Mybatis 提供了 9 种动态 sql 标签：trim | where | set | foreach | if | choose | when | otherwise | bind。

7.12 Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

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。

咨意系Java工程师面试指导课程—samuel

7.13 简述 Mybatis 的插件运行原理，以及如何编写一个插件。

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。

编写插件：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。

7.13.1 Mybatis 是如何进行分页的？分页插件的原理是什么？

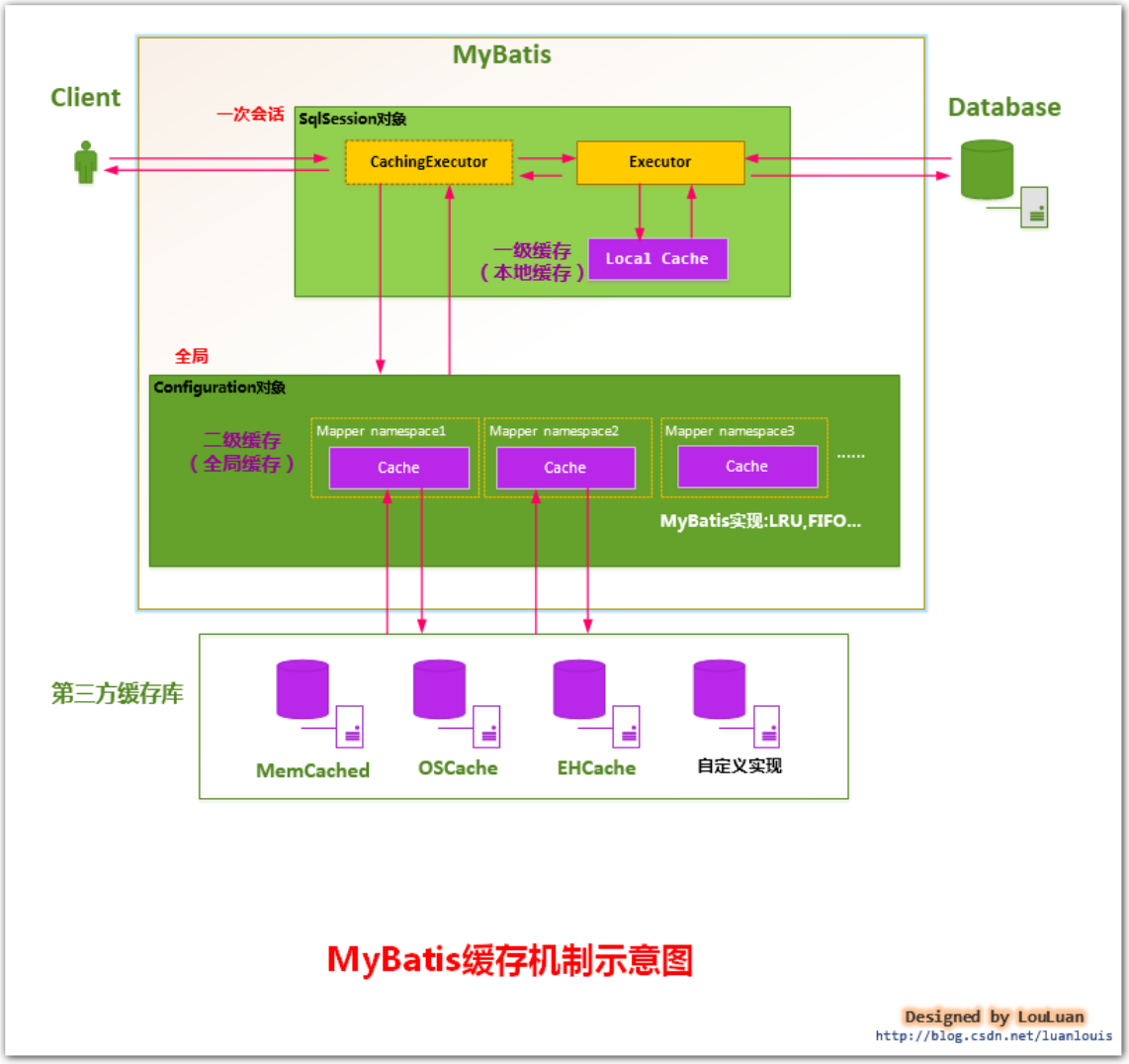
Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页。可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

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。

7.14 Mybatis 缓存

Mybatis 中有一级缓存和二级缓存，默认情况下一级缓存是开启的，而且是不能关闭的。一级缓存 是指 SqlSession 级别的缓存，当在同一个 SqlSession 中进行相同的 SQL 语句查询时，第二次以 后的查询不会从数据库查询，而是直接从缓存中获取，一级缓存最多缓存 1024 条 SQL。二级缓存 是指可以跨 SqlSession 的缓存。是 mapper 级别的缓存，对于 mapper 级别的缓存不同的 sqlsession 是可以共享的。

治愈系Java工程师面试指导课程-samuel



7.14.1 Mybatis 的一级、二级缓存:

1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/> ；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

7.14.2 Mybatis 的一级缓存原理（sqlsession 级别）

第一次发出一个查询 sql，sql 查询结果写入 sqlsession 的一级缓存中，缓存使用的数据结构是一 个 map。
key：MapperID+offset+limit+Sql+所有的入参
value：用户信息
同一个 sqlsession 再次发出相同的 sql，就从缓存中取出数据。如果两次中间出现 commit 操作 （修改、添加、删除），本 sqlsession 中的一级缓存区域全部清空，下次再去缓存中查询不到所 以要从数据库查询，从数据库查询到再写入缓存。

7.14.3 二级缓存原理（mapper 基本）

二级缓存的范围是 mapper 级别（mapper 同一个命名空间），mapper 以命名空间为单位创建缓 存数据结构，结构是 map。mybatis 的二级缓存是通过 CacheExecutor 实现的。CacheExecutor
其实是 Executor 的代理对象。所有的查询操作，在 CacheExecutor 中都会先匹配缓存中是否存 在，不存在则查询数据库。
key：MapperID+offset+limit+Sql+所有的入参
具体使用需要配置：

1. Mybatis 全局配置中启用二级缓存配置
2. 在对应的 Mapper.xml 中配置 cache 节点
3. 在对应的 select 查询节点中添加 useCache=true

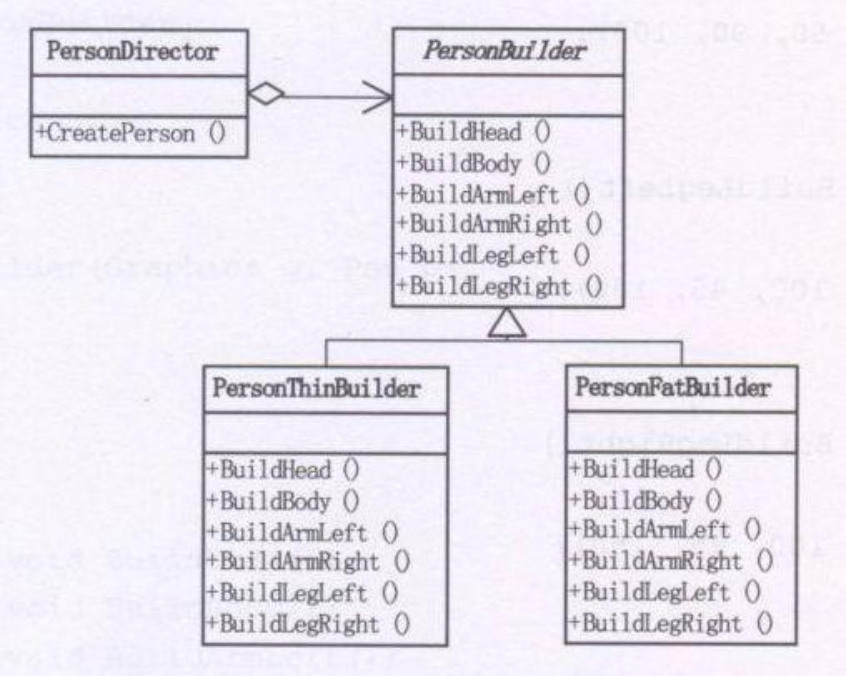
7.15 Mybatis 使用到的设计模式

7.15.1 Builder 模式

例如 SqlSessionFactoryBuilder、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder、CacheBuilder；

Builder 模式的定义是“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。”，它属于创建类模式，一般来说，如果一个对象的构建比较复杂，超出了构造函数所能包含的范围，就可以使用工厂模式和 Builder 模式，相对于工厂模式会产出一个完整的产品，Builder 应用于更加复杂的对象的构建，甚至只会构建产品的一个部分。《effective-java》中第 2 条也提到：遇到多个构造器参数时，考虑用构建者(Builder)模式。

沿愈系Java工程师面试指导课程-samuel



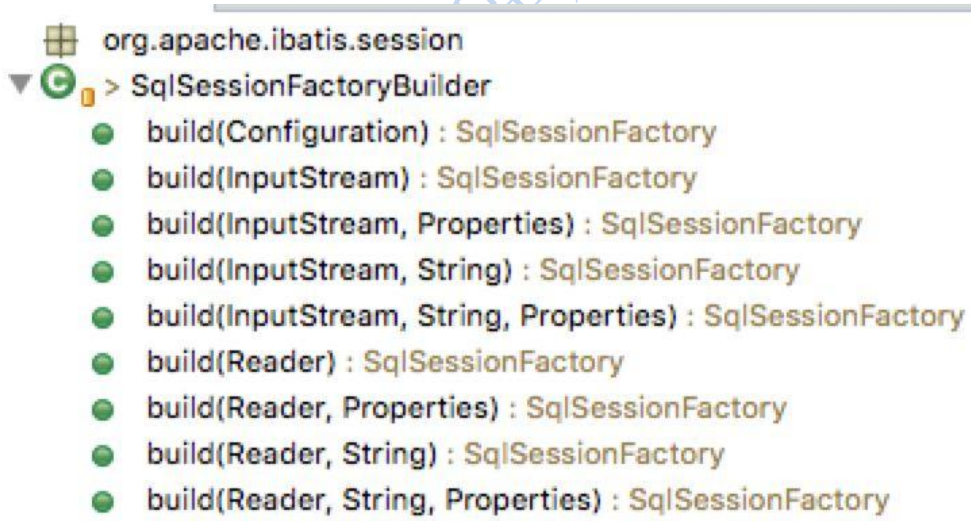
在 Mybatis 环境的初始化过程中，SqlSessionFactoryBuilder 会调用 XMLConfigBuilder 读取所有的 MybatisMapConfig.xml 和所有的*Mapper.xml 文件，构建 Mybatis 运行的核心对象 Configuration 对象，然后将该 Configuration 对象作为参数构建一个 SqlSessionFactory 对象。

其中 XMLConfigBuilder 在构建 Configuration 对象时，也会调用 XMLMapperBuilder 用于读取*.Mapper 文件，而 XMLMapperBuilder 会使用 XMLStatementBuilder 来读取和 build 所有的 SQL 语句。

在这个过程中，有一个相似的特点，就是这些 Builder 会读取文件或者配置，然后做大量的 XpathParser 解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这么多的工作都不是一个构造函数所能包括的，因此大量采用了 Builder 模式来解决。

对于 builder 的具体类，方法都大都用 build*开头，比如 SqlSessionFactoryBuilder 为例，它包含以下方法：

SqlSessionFactoryBuilder

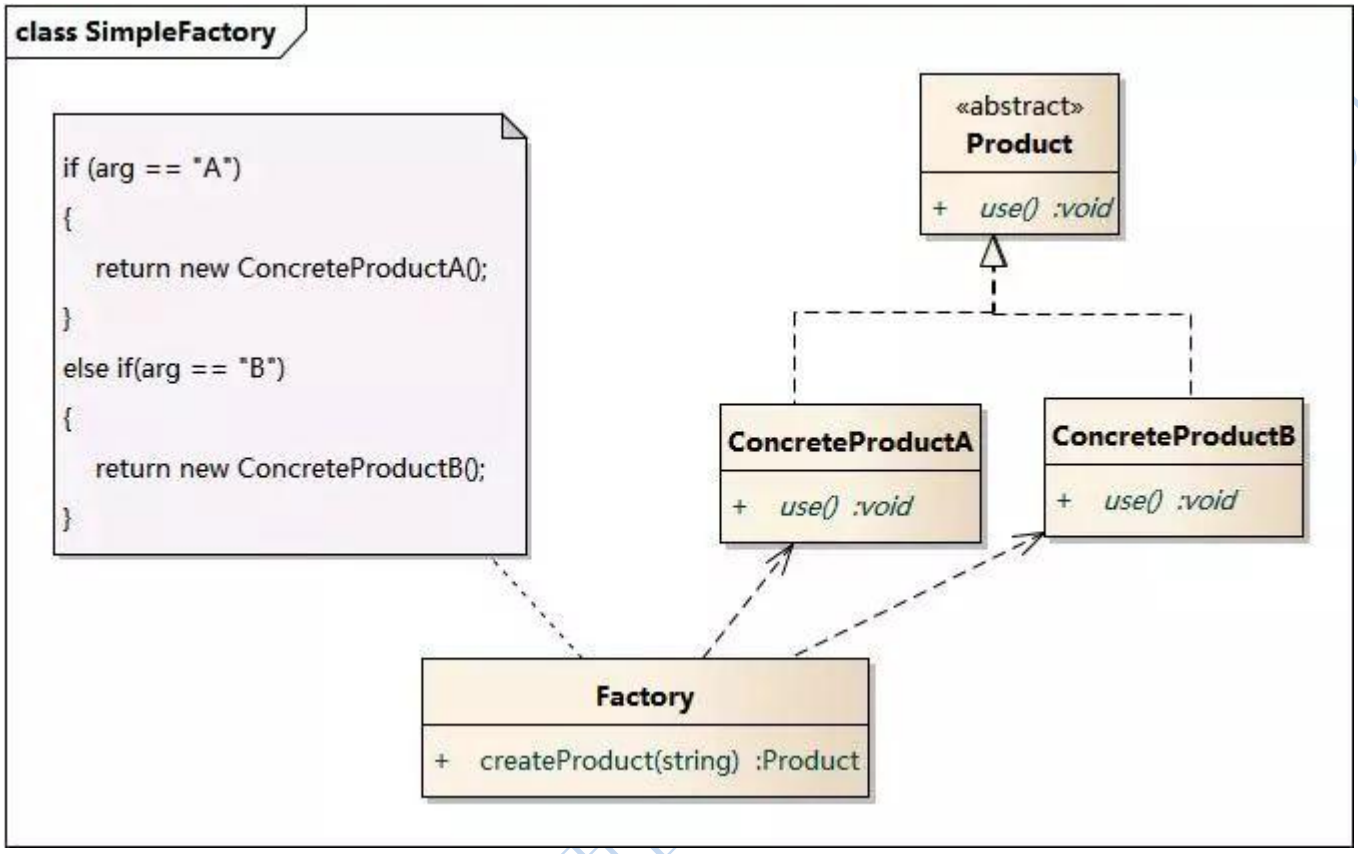


即根据不同的输入参数来构建 SqlSessionFactory 这个工厂对象。

7.15.2 工厂模式

例如 SqlSessionFactory、ObjectFactory、MapperProxyFactory；

在 Mybatis 中比如 SqlSessionFactory 使用的是工厂模式，该工厂没有那么复杂的逻辑，是一个简单工厂模式。简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。



SqlSession 可以认为是一个 Mybatis 工作的核心的接口，通过这个接口可以执行执行 SQL 语句、获取 Mappers、管理事务。类似于连接 MySQL 的 Connection 对象。

可以看到，该 Factory 的 openSession（）方法重载了很多个，分别支持 autoCommit、Executor、Transaction 等参数的输入，来构建核心的 SqlSession 对象。

在 DefaultSqlSessionFactory 的默认工厂实现里，有一个方法可以看出工厂怎么产出一个产品：

```
private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level,
                boolean autoCommit) {
```

```
        Transaction tx = null;
        try {
            final Environment environment = configuration.getEnvironment();
            final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
            tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
            final Executor executor = configuration.newExecutor(tx, execType);
            return new DefaultSqlSession(configuration, executor, autoCommit);
        } catch (Exception e) {
            closeTransaction(tx); // may have fetched a connection so lets call
                                  // close()
            throw ExceptionFactory.wrapException("Error opening session.  Cause: " + e, e);
        } finally {
            ErrorContext.instance().reset();
        }
    }
```

这是一个 openSession 调用的底层方法，该方法先从 configuration 读取对应的环境配置，然后初始化 TransactionFactory 获得一个 Transaction 对象，然后通过 Transaction 获取一个 Executor 对象，最后通过 configuration、Executor、是否 autoCommit 三个参数构建了 SqlSession。

在这里其实也可以看到端倪，SqlSession 的执行，其实是委托给对应的 Executor 来进行的。

而对于 LogFactory，它的实现代码：

```
public final class LogFactory {
    private static Constructor<? extends Log> logConstructor;

    private LogFactory() {
        // disable construction
    }

    public static Log getLog(Class<?> aClass) {
        return getLog(aClass.getName());
    }
```

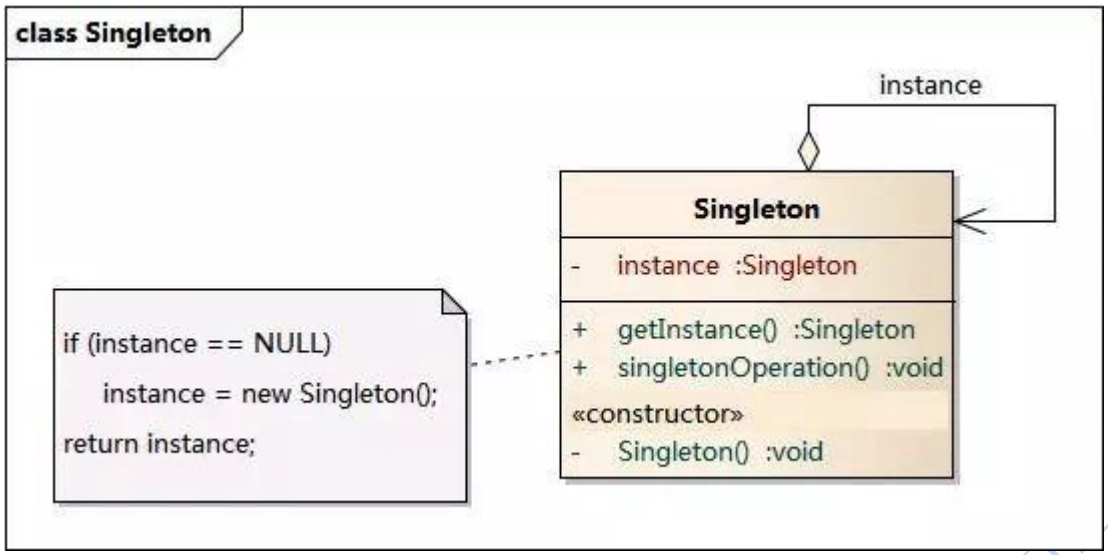
这里有个特别的地方，是 Log 变量的的类型是 Constructor<? extendsLog>，也就是说该工厂生产的不只是一个产品，而是具有 Log 公共接口的一系列产品，比如 Log4jImpl、Slf4jImpl 等很多具体的 Log。

7.15.3 单例模式

例如 ErrorContext 和 LogFactory；
单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

泊鱼系Java工程师面试指导课程——samuel



在 Mybatis 中有两个地方用到单例模式，ErrorContext 和 LogFactory，其中 ErrorContext 是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而 LogFactory 则是提供给整个 Mybatis 使用的日志工厂，用于获得针对项目配置好的日志对象。

ErrorContext 的单例实现代码：

```
public class ErrorContext {

    private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>();

    private ErrorContext() {
    }

    public static ErrorContext instance() {
        ErrorContext context = LOCAL.get();
        if (context == null) {
            context = new ErrorContext();
            LOCAL.set(context);
        }
        return context;
    }
```

构造函数是 private 修饰，具有一个 static 的局部 instance 变量和一个获取 instance 变量的方法，在获取实例的方法中，先判断是否为空如果是的话就先创建，然后返回构造好的对象。

只是这里有个有趣的地方是，LOCAL 的静态实例变量使用了 ThreadLocal 修饰，也就是说它属于每个线程各自的数据，而在 instance()方法中，先获取本线程的该实例，如果没有就创建该线程独有的 ErrorContext。

7.15.4 代理模式

Mybatis 实现的核心，比如 MapperProxy、ConnectionLogger，用的 jdk 的动态代理；还有 executor.loader 包使用了 cglib 或者 javassist 达到延迟加载的效果；

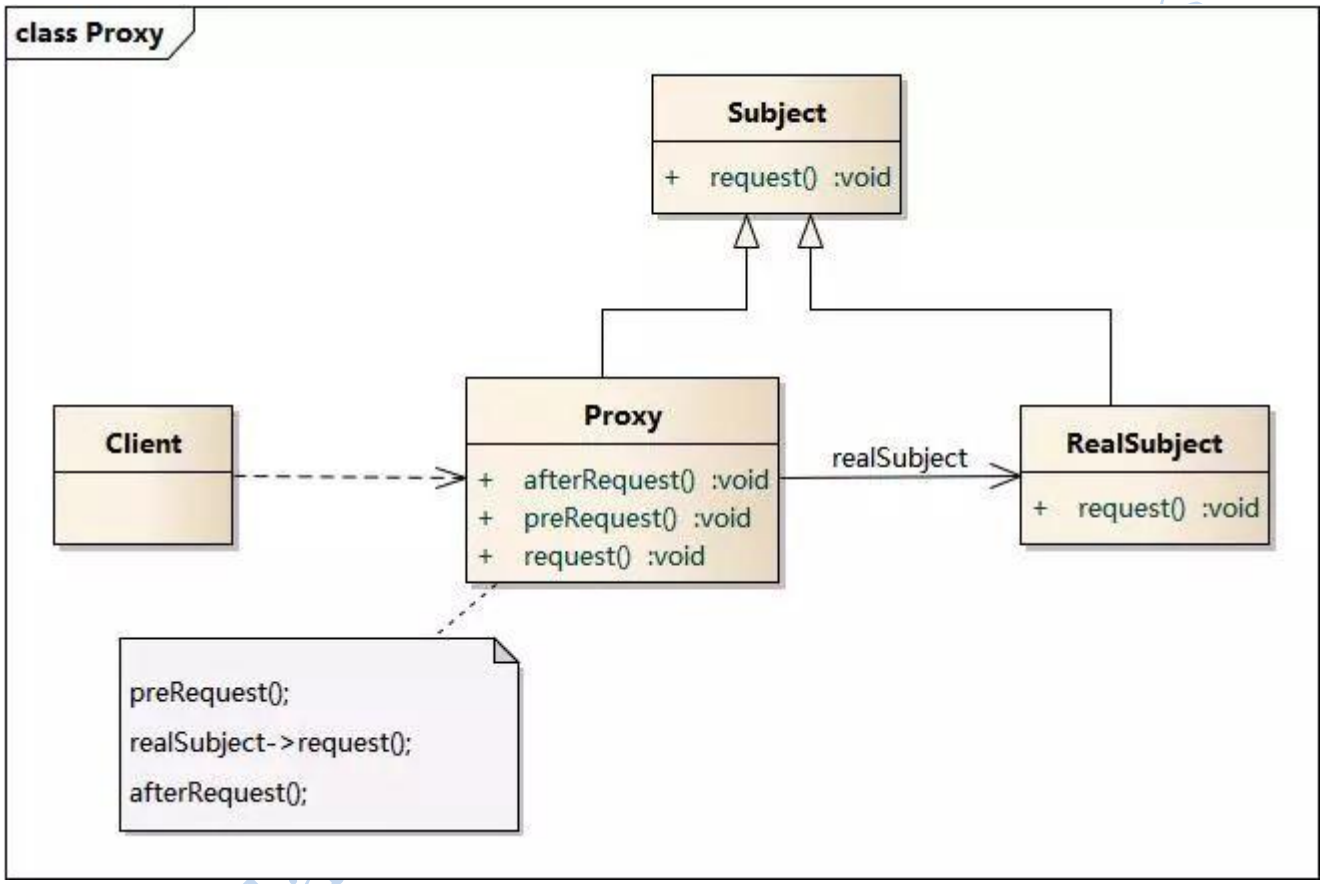
咨询乔Java工程师面试指导课程--samuel

代理模式可以认为是 Mybatis 的核心使用的模式，正是由于这个模式，我们只需要编写 Mapper.java 接口，不需要实现，由 Mybatis 后台帮我们完成具体 SQL 的执行。

代理模式(Proxy Pattern) ：给某一个对象提供一个代 理，并由代理对象控制对原对象的引用。代理模式的英 文叫做 Proxy 或 Surrogate，它是一种对象结构型模式。

代理模式包含如下角色：

Subject: 抽象主题角色
Proxy: 代理主题角色
RealSubject: 真实主题角色



这里有两个步骤，第一个是提前创建一个 Proxy，第二个是使用的时候会自动请求 Proxy，然后由 Proxy 来执行具体事务；

当我们使用 Configuration 的 getMapper 方法时，会调用 mapperRegistry.getMapper 方法，而该方法又会调用 mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

```
/**
 * @author Lasse Voss
 */
public class MapperProxyFactory<T> {

    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();
```

```
    public MapperProxyFactory(Class<T> mapperInterface) {
        this.mapperInterface = mapperInterface;
    }

    public Class<T> getMapperInterface() {
        return mapperInterface;
    }

    public Map<Method, MapperMethod> getMethodCache() {
        return methodCache;
    }

    @SuppressWarnings("unchecked")
    protected T newInstance(MapperProxy<T> mapperProxy) {
        return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface },
                mapperProxy);
    }

    public T newInstance(SqlSession sqlSession) {
        final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);
        return newInstance(mapperProxy);
    }

}
```

在这里，先通过 T newInstance(SqlSession sqlSession)方法会得到一个 MapperProxy 对象，然后调用 T newInstance(MapperProxy<T> mapperProxy)生成代理对象然后返回。

而查看 MapperProxy 的代码，可以看到如下内容：

```
public class MapperProxy<T> implements InvocationHandler, Serializable {

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            if (Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);
            } else if (isDefaultMethod(method)) {
                return invokeDefaultMethod(proxy, method, args);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
        final MapperMethod mapperMethod = cachedMapperMethod(method);
        return mapperMethod.execute(sqlSession, args);
    }
```

治愈系Java工程师面试指导课程——samuel

非常典型的，该 MapperProxy 类实现了 InvocationHandler 接口，并且实现了该接口的 invoke 方法。

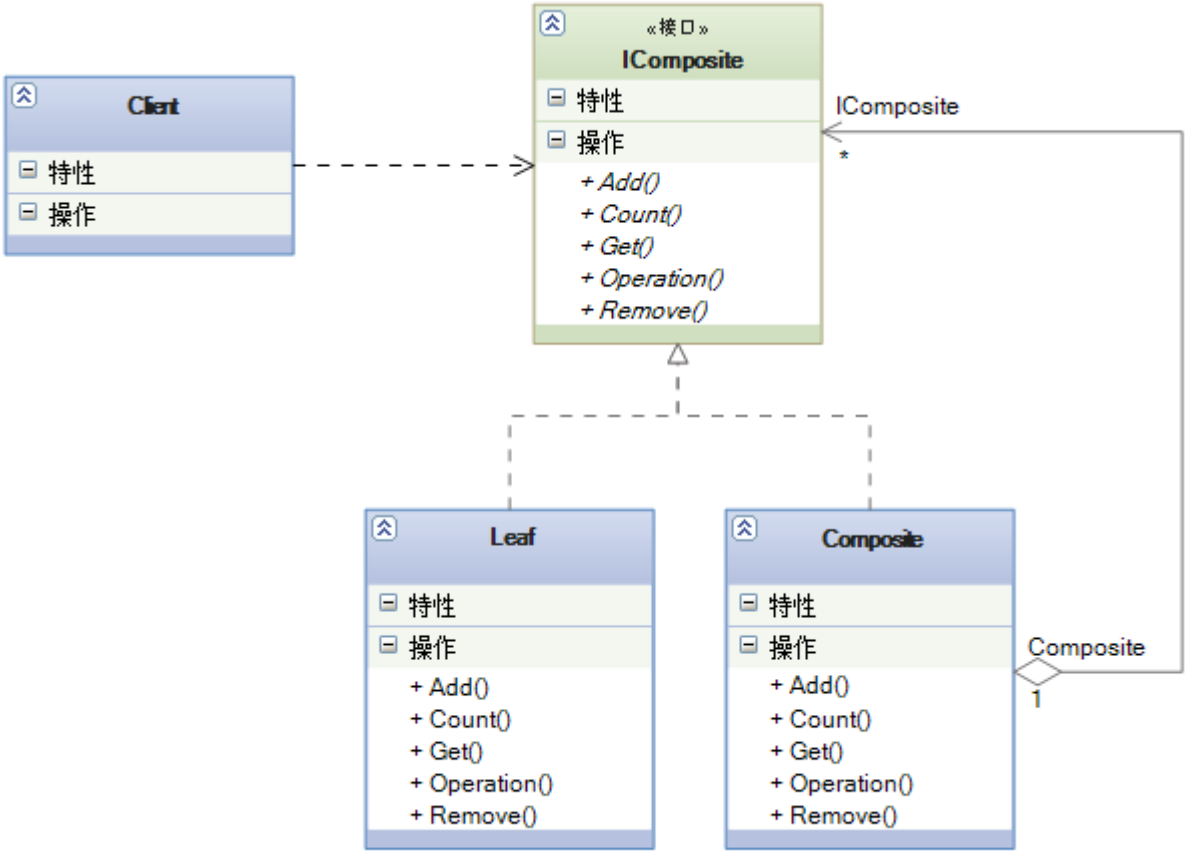
通过这种方式，我们只需要编写 Mapper.java 接口类，当真正执行一个 Mapper 接口的时候，就会转发给 MapperProxy.invoke 方法，而该方法则会调用后续的 sqlSession.cud>executor.execute>prepareStatement 等一系列方法，完成 SQL 的执行和返回。

7.15.5 组合模式

例如 SqlNode 和各个子类 ChooseSqlNode 等；
组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。

组合模式对单个对象(叶子对象)和组合对象(组合对象)具有一致性，它将对象组织到树结构中，可以用来描述整体与部分的关系。同时它也模糊了简单元素(叶子对象)和复杂元素(容器对象)的概念，使得客户能够像处理简单元素一样来处理复杂元素，从而使客户程序能够与复杂元素的内部结构解耦。

在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。



Mybatis 支持动态 SQL 的强大功能，比如下面的这个 SQL：

```
<update id="update" parameterType="org.format.dynamicproxy.mybatis.bean.User">
    UPDATE users
    <trim prefix="SET" prefixOverrides=",">
        <if test="name != null and name != ''">
```

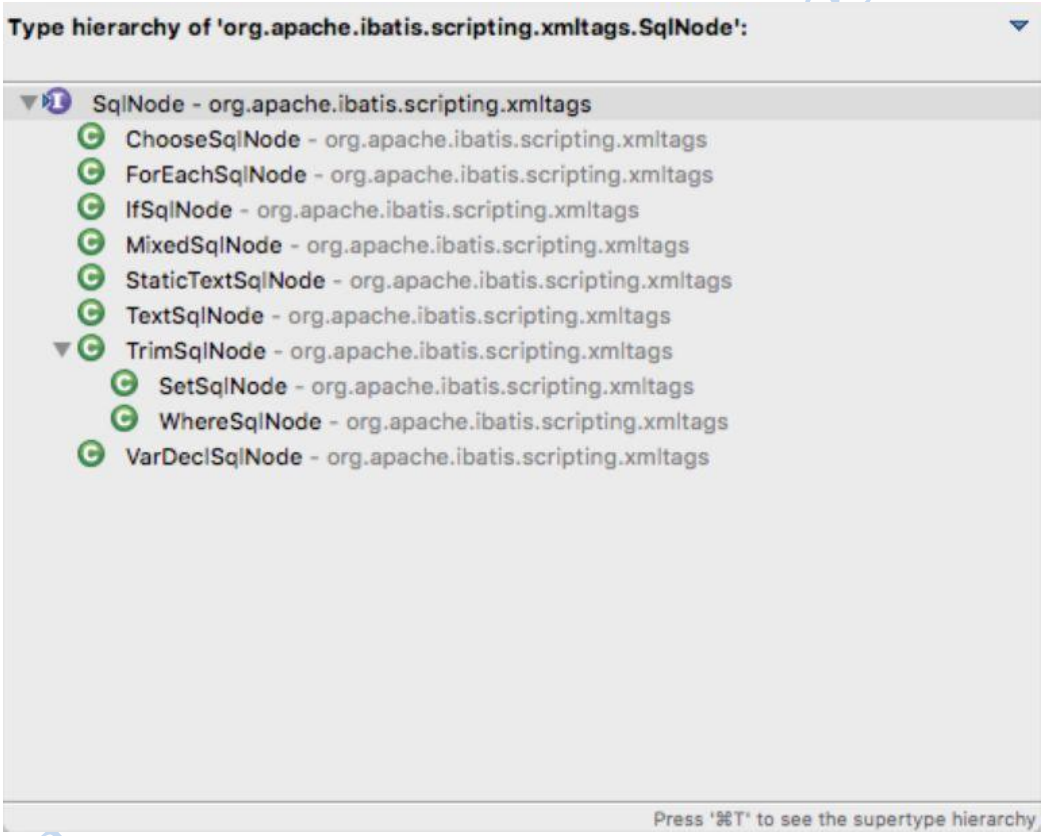
```
            name = #{name}
        </if>
        <if test="age != null and age != ''">
            , age = #{age}
        </if>
        <if test="birthday != null and birthday != ''">
            , birthday = #{birthday}
        </if>
    </trim>
    where id = ${id}
</update>
```

在这里面使用到了 trim、if 等动态元素，可以根据条件来生成不同情况下的 SQL；

在 DynamicSqlSource.getBoundSql 方法里，调用了 rootSqlNode.apply(context)方法，apply 方法是所有的动态节点都实现的接口：

```
public interface SqlNode {
    boolean apply(DynamicContext context);
}
```

对于实现该 SqlSource 接口的所有节点，就是整个组合模式树的各个节点：



组合模式的简单之处在于，所有的子节点都是同一类节点，可以递归的向下执行，比如对于 TextSqlNode，因为它是最底层的叶子节点，所以直接将对应的内容 append 到 SQL 语句中：

```
    @Override
    public boolean apply(DynamicContext context) {
        GenericTokenParser parser = createParser(new BindingTokenParser(context, injectionFilter));
        context.appendSql(parser.parse(text));
```

```
            return true;
        }
```

但是对于 IfSqlNode，就需要先做判断，如果判断通过，仍然会调用子元素的 SqlNode，即 contents.apply 方法，实现递归的解析。

```
@Override
public boolean apply(DynamicContext context) {
    if (evaluator.evaluateBoolean(test, context.getBindings())) {
        contents.apply(context);
        return true;
    }
    return false;
}
```

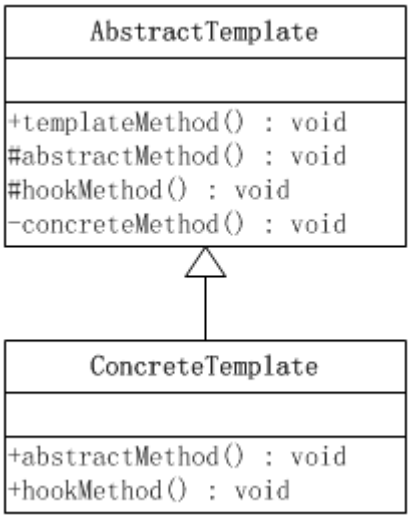
7.15.6 模板方法模式

例如 BaseExecutor 和 SimpleExecutor，还有 BaseTypeHandler 和所有的子类例如 IntegerTypeHandler；

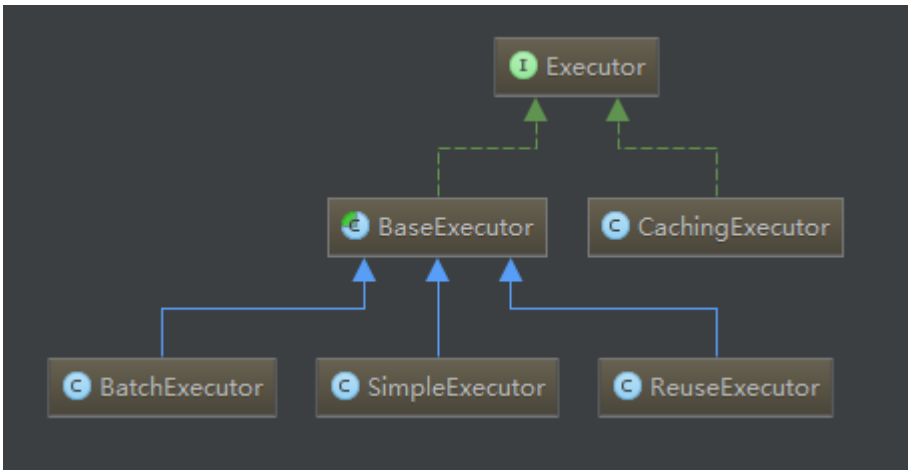
模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method)；而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是从此而来。

模板类定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。



在 Mybatis 中，sqlSession 的 SQL 执行，都是委托给 Executor 实现的，Executor 包含以下结构：



其中的 BaseExecutor 就采用了模板方法模式，它实现了大部分的 SQL 执行逻辑，然后把以下几个方法交给子类定制化完成：

```
  @Override
    public boolean apply(DynamicContext context) {
        if (evaluator.evaluateBoolean(test, context.getBindings())) {
            contents.apply(context);
            return true;
        }
        return false;
    }
```

该模板方法类有几个子类的具体实现，使用了不同的策略：

简单 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。（可以是 Statement 或 PrepareStatement 对象）
重用 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，用完后，不关闭 Statement 对象，而是放置于 Map 内，供下一次使用。（可以是 Statement 或 PrepareStatement 对象）
批量 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()批处理的；BatchExecutor 相当于维护了多个桶，每个桶里都装了很多属于自己的 SQL，就像苹果蓝里装了很多苹果，番茄蓝里装了很多番茄，最后，再统一倒进仓库。（可以是 Statement 或 PrepareStatement 对象）
比如在 SimpleExecutor 中这样实现 update 方法：

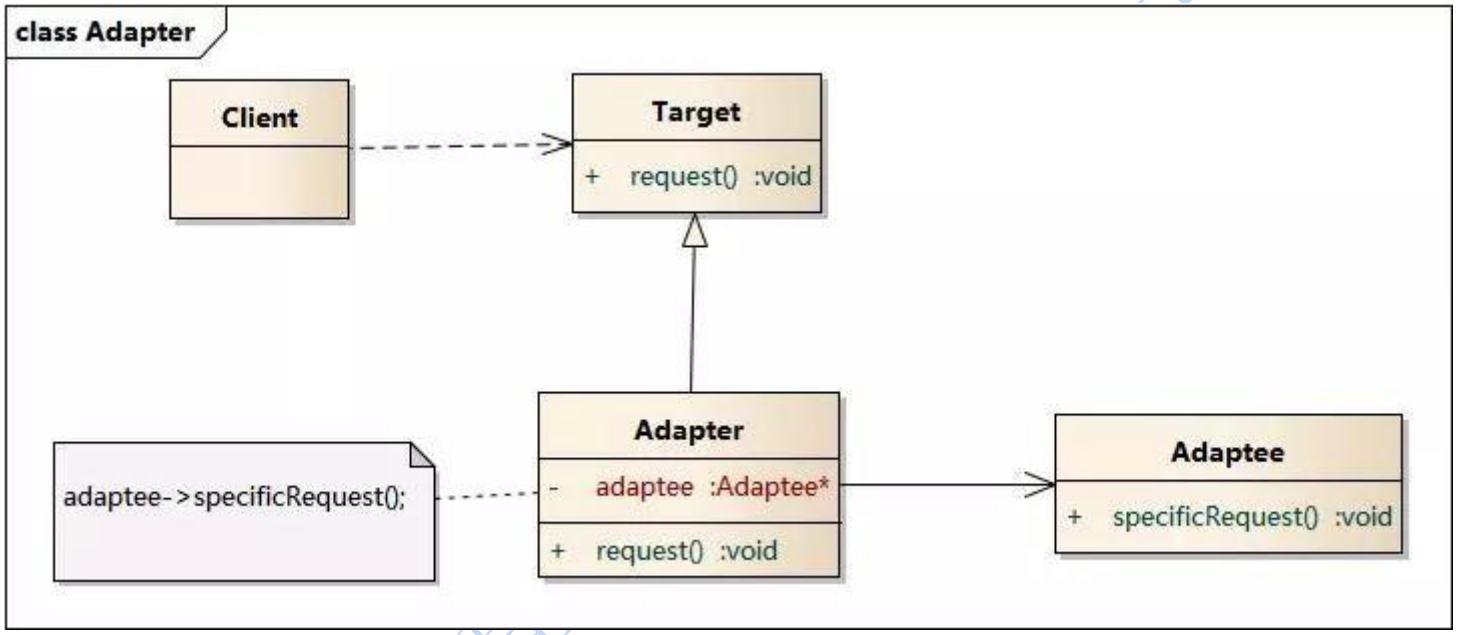
```
    @Override
    public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
        Statement stmt = null;
        try {
            Configuration configuration = ms.getConfiguration();
            StatementHandler handler = configuration.newStatementHandler(this, ms, parameter,
RowBounds.DEFAULT, null,
                    null);
            stmt = prepareStatement(handler, ms.getStatementLog());
            return handler.update(stmt);
        } finally {
```

沿途学Java工程师面试指导课程--samuel

```
            closeStatement(stmt);
        }
    }
```

7.15.7 适配器模式

例如 Log 的 Mybatis 接口和它对 jdbc、log4j 等各种日志框架的适配实现；
适配器模式(Adapter Pattern) ：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

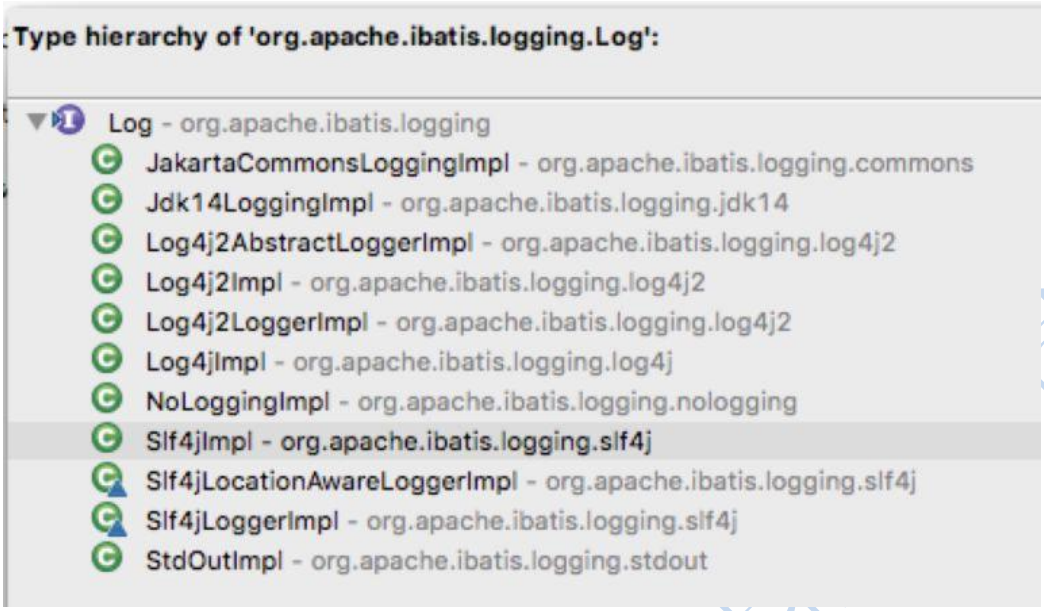


在 Mybatsi 的 logging 包中，有一个 Log 接口：

```
/**
 * @author Clinton Begin
 */
public interface Log {

    boolean isDebugEnabled();

    boolean isTraceEnabled();

    void error(String s, Throwable e);

    void error(String s);

    void debug(String s);

    void trace(String s);
```

```
    void warn(String s);

}
```

该接口定义了 Mybatis 直接使用的日志方法，而 Log 接口具体由谁来实现呢？Mybatis 提供了多种日志框架的实现，这些实现都匹配这个 Log 接口所定义的接口方法，最终实现了所有外部日志框架到 Mybatis 日志包的适配：



比如对于 Log4jImpl 的实现来说，该实现持有了 org.apache.log4j.Logger 的实例，然后所有的日志方法，均委托该实例来实现。

```
public class Log4jImpl implements Log {

    private static final String FQCN = Log4jImpl.class.getName();

    private Logger log;

    public Log4jImpl(String clazz) {
        log = Logger.getLogger(clazz);
    }

    @Override
    public boolean isDebugEnabled() {
        return log.isDebugEnabled();
    }

    @Override
    public boolean isTraceEnabled() {
        return log.isTraceEnabled();
    }

    @Override
    public void error(String s, Throwable e) {
```

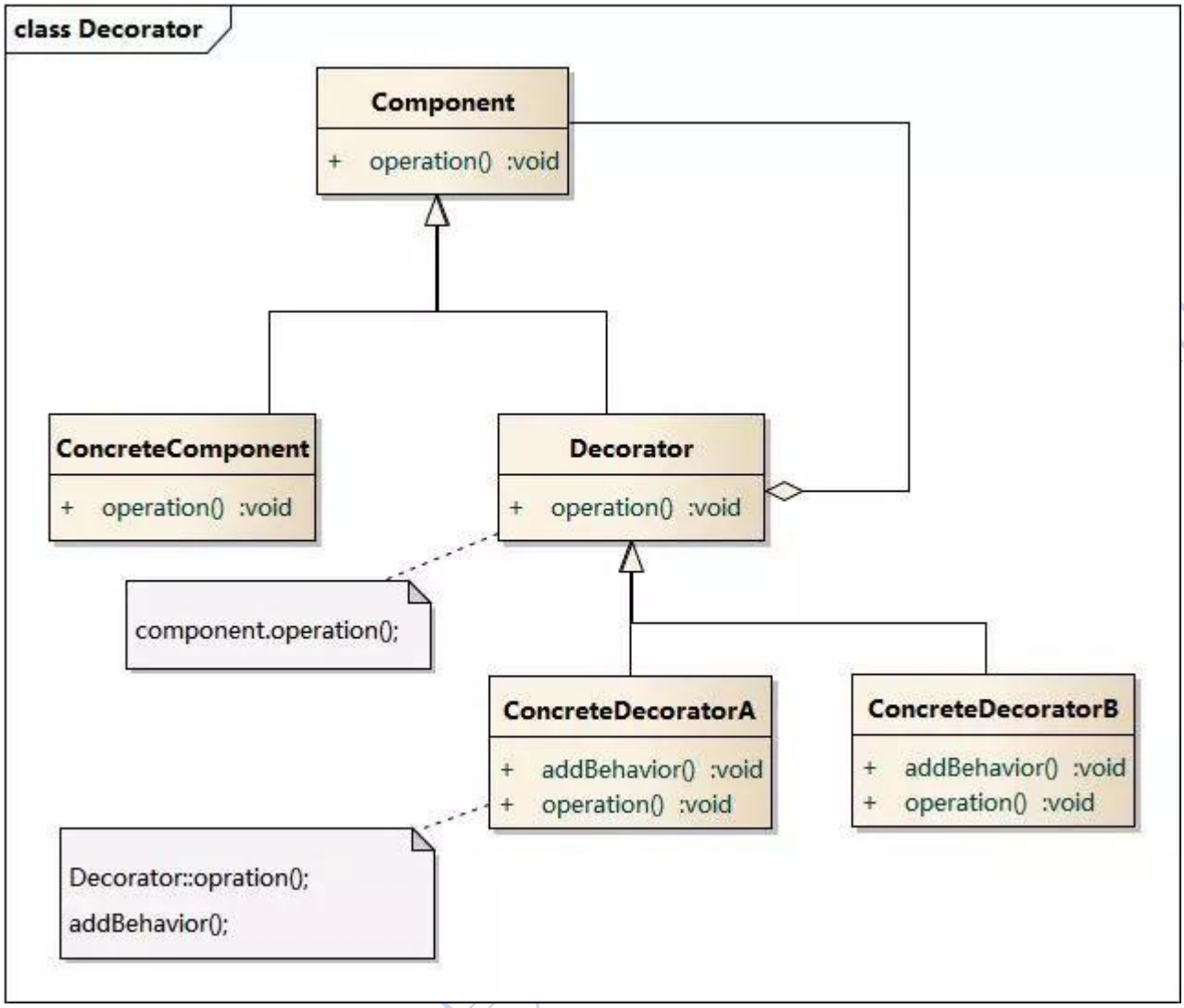
治愈系Java工程师面试宝典——samuel

```
            log.log(FQCN, Level.ERROR, s, e);
        }

        @Override
        public void error(String s) {
            log.log(FQCN, Level.ERROR, s, null);
        }

        @Override
        public void debug(String s) {
            log.log(FQCN, Level.DEBUG, s, null);
        }

        @Override
        public void trace(String s) {
            log.log(FQCN, Level.TRACE, s, null);
        }

        @Override
        public void warn(String s) {
            log.log(FQCN, Level.WARN, s, null);
        }

}
```

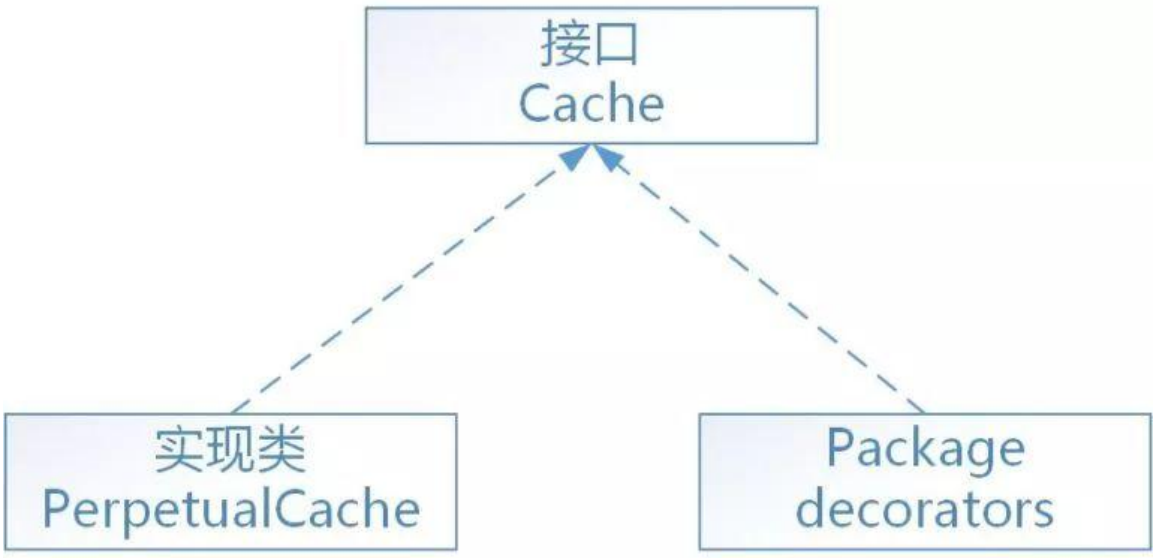
7.15.8 装饰者模式

例如 cache 包中的 cache.decorators 子包中等各个装饰者的实现；
装饰模式(Decorator Pattern) ：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

治愈系Java工程师面试指导课程——samuel



在 mybatis 中，缓存的功能由根接口 Cache（org.apache.ibatis.cache.Cache）定义。整个体系采用装饰器设计模式，数据存储和缓存的基本功能由 PerpetualCache（org.apache.ibatis.cache.impl.PerpetualCache）永久缓存实现，然后通过一系列的装饰器来对 PerpetualCache 永久缓存进行缓存策略等方便的控制。如下图：



用于装饰 PerpetualCache 的标准装饰器共有 8 个（全部在 org.apache.ibatis.cache.decorators 包中）：

FifoCache：先进先出算法，缓存回收策略
LoggingCache：输出缓存命中的日志信息
LruCache：最近最少使用算法，缓存回收策略
ScheduledCache：调度缓存，负责定时清空缓存
SerializedCache：缓存序列化和反序列化存储
SoftCache：基于软引用实现的缓存管理策略
SynchronizedCache：同步的缓存装饰器，用于防止多线程并发访问
WeakCache：基于弱引用实现的缓存管理策略
另外，还有一个特殊的装饰器 TransactionalCache：事务性的缓存

正如大多数持久层框架一样，mybatis 缓存同样分为一级缓存和二级缓存

一级缓存，又叫本地缓存，是 PerpetualCache 类型的永久缓存，保存在执行器中（BaseExecutor），而执行器又在 SqlSession（DefaultSqlSession）中，所以一级缓存的生命周期与 SqlSession 是相同的。
二级缓存，又叫自定义缓存，实现了 Cache 接口的类都可以作为二级缓存，所以可配置如 encache 等的第三方缓存。
二级缓存以 namespace 名称空间为其唯一标识，被保存在 Configuration 核心配置对象中。
二级缓存对象的默认类型为 PerpetualCache，如果配置的缓存是默认类型，则 mybatis 会根据配置自动追加一系列装饰器。

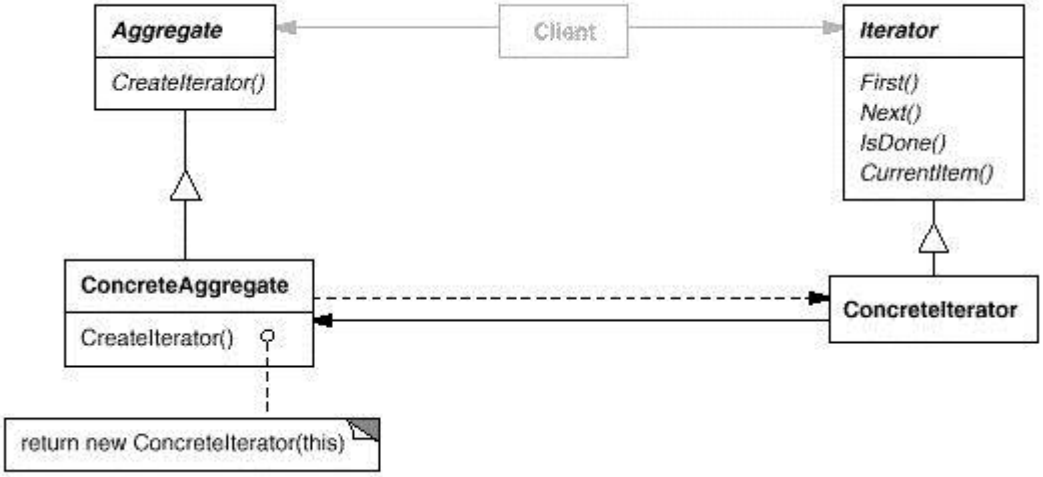
Cache 对象之间的引用顺序为：

SynchronizedCache–>LoggingCache–>SerializedCache–>ScheduledCache–>LruCache–>PerpetualCache

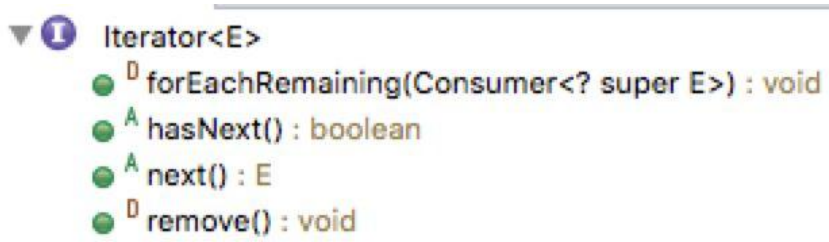
7.15.9 迭代器模式

例如迭代器模式 PropertyTokenizer；
接下来挨个模式进行解读，先介绍模式自身的知识，然后解读在 Mybatis 中怎样应用了该模式。
迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF 给出的定义为：提供一种方法访问一个容器（container）对象中各个元素，而又不需暴露该对象的内部细节。

治愈系JAVA工程师面试指导课程--samuel



Java 的 Iterator 就是迭代器模式的接口，只要实现了该接口，就相当于应用了迭代器模式：



比如 Mybatis 的 PropertyTokenizer 是 property 包中的重量级类，该类会被 reflection 包中其他的类频繁的引用到。这个类实现了 Iterator 接口，在使用时经常被用到的是 Iterator 接口中的 hasNext 这个函数。

```
public class PropertyTokenizer implements Iterator<PropertyTokenizer> {
    private String name;
    private String indexedName;
    private String index;
    private String children;

    public PropertyTokenizer(String fullname) {
        int delim = fullname.indexOf('.');
        if (delim > -1) {
            name = fullname.substring(0, delim);
            children = fullname.substring(delim + 1);
        } else {
            name = fullname;
            children = null;
        }
        indexedName = name;
        delim = name.indexOf('[');
        if (delim > -1) {
            index = name.substring(delim + 1, name.length() - 1);
            name = name.substring(0, delim);
        }
```

```
    }

    public String getName() {
        return name;
    }

    public String getIndex() {
        return index;
    }

    public String getIndexedName() {
        return indexedName;
    }

    public String getChildren() {
        return children;
    }

    @Override
    public boolean hasNext() {
        return children != null;
    }

    @Override
    public PropertyTokenizer next() {
        return new PropertyTokenizer(children);
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException(
                "Remove is not supported, as it has no meaning in the context of properties.");
    }
}
```

可以看到，这个类传入一个字符串到构造函数，然后提供了 iterator 方法对解析后的子串进行遍历，是一个很常用的方法类。

治愈系Java工程师面试指导课程——samuel

8 微服务概念

8.1 什么是微服务

微服务的架构理念，是将我们原本庞大的业务细化为一个一个小的业务单元，其中每个小型服务都运行在自己的进

程中，并经常采用 HTTP 资源 API 这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术。每一个微服务都有自己的业务逻辑和适配器。一些微服务还会发布 API 给其它微服务和应用客户端使用，运行时，每一个实例可能是一个云 VM 或者是 Docker 容器，他们集合到一起对外提供一个整体大型应用的服务

8.2 微服务和传统服务相比，有什么好处呢

易于开发和维护 (基于 Springboot 配置少 开发框)
启动较快
局部修改容易部署 （使用 docker 容器打包镜像 非常封边）
技术栈不受限 （不同的服务间比较独立）
按需伸缩 (来业务了就加服务，业务部挣钱了就砍服务)
DevOps (全自动、持续集成的部署理念 -> docker jenkens maven rancher)

8.3 什么是 restful 风格

restful 是软件 API 接口的一种设计风格，现在前后台分离、以及前端的各种移动设备的流行使得前后端分离架构更加的流行，
前台和后台的交互更是变得愈加重要，而 restful 风格就是该对如何进行 web 接口的设计进行了设计，让我们的 Web 接口更加统一和规范。
具体来说：
1）每一个 URI 代表一种资源；
访问用户 /user
访问日志 /log
（2）客户端通过四个 HTTP 动词，对服务器端资源进行操作，实现"表现层状态转化"。
通过请求的 method 代表 这次请求要做什么 常用的 GET 请求获取 POST 添加 PUT 修改 DELETE 删除
（3）需要单个主键类的，如获取 ID 为 1 的用户 都是使用路径传参的格式传递 ID
查询 http://localhost:8080/user/1 method=GET
删除 http://localhost:8080/user/1 method=DELETE

8.4 SpringBoot、SpringCloud 及微服务三者间的区别和联系

SpringBoot 是一个脚手架项目，主要是用特有的方式来快速的配置和启动基于 Spring 的项目，让我们开发基于 Spring 的企业级应用变得更加简单。
微服务呢上面有说 ，是一种架构理念。
只要是架构就得有人来实现，而 SpringCloud 提供了大量的组件实现了一站式的微服务解决方案。
而 SpringCloud 开发的过程当中是基于 SpringBoot 开发的。

沧海系Java工程师面试指导课程——samuel

9 SpringBoot

9.1 什么是 Spring Boot?

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式处理方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。它可用于创建可执行的 Spring 应用程序，采用了习惯优于配置的方法。

9.2 为什么要用 Spring Boot?

Spring Boot 优点非常多，如：

- 独立运行
- 简化配置
- 自动配置
- 无代码生成和 XML 配置
- 应用监控
- 上手容易

…

9.3 Spring Boot 特点

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭 建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的 配置。通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。其特点如下：

1. 创建独立的 Spring 应用程序
2. 嵌入的 Tomcat，无需部署 WAR 文件
3. 简化 Maven 配置
4. 自动配置 Spring
5. 提供生产就绪型功能，如指标，健康检查和外部配置
6. 绝对没有代码生成和对 XML 没有要求配置

9.4 SpringBoot 的配置文件

9.4.1 Spring Boot 的核心配置文件

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。
application 配置文件这个容易了解，主要用于 Spring Boot 项目的自动化配置。
bootstrap 配置文件有以下几个应用场景。

沿途教育Java工程师面试指导课程——samuel

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中增加连接到配置中心的配置属性来加载外部配置中心的配置信息；
少量固定的不能被覆盖的属性；
少量加密/解密的场景；

9.4.2 Spring Boot 有哪几种读取配置的方式？

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量，

9.4.3 Spring Boot 的配置文件有哪几种格式？它们有什么区别？

.properties 和 .yml，它们的区别主要是书写格式不同。

1).properties

```
app.user.name = javastack
```

2).yml

```
app:
  user:
    name: javastack
```

另外，.yml 格式不支持@PropertySource 注解导入配置。

9.4.4 你如何了解 Spring Boot 配置加载顺序？

在 Spring Boot 里面，可以使用以下几种方式来加载配置。
1）properties 文件；
2）YAML 文件；
3）系统环境变量；
4）命令行参数；
等等……

9.4.5 Spring Boot 如何定义多套不同环境配置？

提供多套配置文件，如：

治愈系Java工程师面试指导课程——samuel

运行时通过命令行-P 参数指定具体的配置文件借助 maven 的 profiles

9.5 运行 Spring Boot 有哪几种方式？

1）打包用命令或者者放到容器中运行
2）用 Maven/ Gradle 插件运行
3）直接执行 main 方法运行

9.6 SpringBoot 其他的常用注解

- @Enable*注释 @Enable*注释并不是新发明的注释，早在 Spring 3 框架就引入了这些注释，用这些注释替代 XML 配置文件。 很多 Spring 开发者都知道@EnableTransactionManagement 注释，它能够声明事务管理；@EnableWebMvc 注释，它能启用 Spring MVC；以及@EnableScheduling 注释，它可以初始化一个调度器。

- ConfigurationProperties 属性映射 下面看 MongoProperties 类，它是一个 Spring Boot 属性映射的例子：

```
@ConfigurationProperties(prefix = "spring.data.mongodb")
 public class MongoProperties {
     private String host;
     private int port = DBPort.PORT;
     private String uri = "mongodb://localhost/test";
     private String database;
     // ... getters setters omitted
 }
```

@ConfigurationProperties 注释将 POJO 关联到指定前缀的每一个属性。例如，spring.data.mongodb.port 属性将映射到这个类的端口属性。 强烈建议 Spring Boot 开发者使用这种方式来删除与配置属性相关的瓶颈代码。

- @Conditional 注释 Spring Boot 的强大之处在于使用了 Spring 4 框架的新特性：@Conditional 注释，此注释使得只有在特定条件满足时才启用一些配置。 在 Spring Boot 的 org.springframework.boot.autoconfigure.condition 包中说明了使用@Conditional 注释能给我们带来什么，下面对这些注释做一个概述： @ConditionalOnBean @ConditionalOnClass @ConditionalOnExpression @ConditionalOnMissingBean @ConditionalOnMissingClass @ConditionalOnNotWebApplication @ConditionalOnResource @ConditionalOnWebApplication

治愈系Java工程师面试指导课程——samuel

9.7 Spring Boot 自动配置原理

Spring Boot 的开启注解是：@SpringBootApplication，其实它就是由下面三个注解组成的：

- @Configuration 注解，实现配置文件的功能。Spring 自带的，和 Spring Boot 无关
- @ComponentScan：Spring 组件扫描。
- @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项,SpringBoot 框架的神奇之处在于@EnableAutoConfiguration 注解，此注释自动载入应用程序所需的所有 Bean——这依赖于 Spring Boot 在类路径中的查找。 它能根据类路径下的 jar 包和配置，动态加载配置和注入 bean。注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下能否有这个类去自动配置。
  比如我在 lib 下放一个 druid 连接池的 jar 包，然后在 application.yml 文件配置 druid 相关的参数，Spring Boot 就能够自动配置所有我们需要的东西，如果我把 jar 包拿掉或者把参数去掉，那 Spring Boot 就不会自动配置。

9.8 SpringBoot 的 Starters

Starters 可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其余技术，而不需要四处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只需加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。
Starters 包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

9.8.1 常见的 starter 会包几个方面的内容。分别是什么。

常见的 starter 会包括下面四个方面的内容

- 自动配置文件，根据 classpath 是否存在指定的类来决定是否要执行该功能的自动配置。
- spring.factories，非常重要，指导 Spring Boot 找到指定的自动配置文件。
- endpoint：可以理解为一个 admin，包含对服务的描述、界面、交互(业务信息的查询)。
- health indicator:该 starter 提供的服务的健康指标。

两个需要注意的点：

1. @ConditionalOnMissingBean 的作用是：只有对应的 bean 在系统中都没有被创建，它修饰的初始化代码块才会执行，【用户自己手动创建的 bean 优先】。

2. Spring Boot Starter 找到自动配置文件(xxxxAutoConfiguration 之类的文件)的方式有两种：spring.factories:由 Spring Boot 触发探测 classpath 目录下的类，进行自动配置；
@EnableXxxxx:有时需要由 starter 的用户触发*查找自动配置文件的过程

9.8.2 Spring Boot Starter 的工作原理

Spring Boot Starter 的工作原理如下：
1. Spring Boot 在启动时扫描项目所依赖的 JAR 包，寻找包含 spring.factories 文件的 JAR

沿途教Java工程师面试指导课程 - samuel

2. 根据 spring.factories 配置加载 AutoConfigure 类
3. 根据 @Conditional 注解的条件，进行自动配置并将 Bean 注入 Spring Context

9.8.3 自定义 springboot-starter 注意事项

1. springboot 默认 scan 的包名是其 main 类所在的包名。如果引入的 starter 包名不一样，需要自己添加 scan。@ComponentScan(basePackages = {"com.xixicat.demo","com.xixicat.sms"})
2. 对于 starter 中有 feign 的，需要额外指定 @EnableFeignClients(basePackages = {"com.xixicat.sms"})
3. 对于 exclude 一些 autoConfig @EnableAutoConfiguration(exclude ={MetricFilterAutoConfiguration.class})

9.9 Spring Boot 支持哪些日志框架？推荐和默认的日志框架是哪个？

Spring Boot 支持 Java Util Logging, Log4j2, Lockback 作为日志框架，假如你使用 Starters 启动器，Spring Boot 将使用 Logback 作为默认日志框架，

9.10 SpringBoot 实现热部署有哪几种方式？

主要有两种方式：
Spring Loaded
Spring-boot-devtools
Spring-boot-devtools?

9.11 Spring Boot 可以兼容老 Spring 项目吗，如何做？

可以兼容，使用@ImportResource 注解导入老 Spring 项目配置文件。

9.12 保护 Spring Boot 应用有哪些方法？

使用 Snyk 检查你的依赖关系
更新到最新版本
启用 CSRF 保护
使用内容安全策略防止 XSS 攻击
在生产中使用 HTTPS
…

价值千金Java工程师面试指导课程——samuel

9.13 Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？

配置变更
JDK 版本更新
第三方类库更新
响应式 Spring 编程支持
HTTP/2 支持
配置属性绑定
更多改进与增强…

9.14 Spring Boot、Spring MVC 和 Spring 有什么区别？

- SpringFramework 最重要的特征是依赖注入。所有 SpringModules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

- Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。
  Spring 和 SpringMVC 的问题在于需要配置大量的参数。
- Spring Boot 通过一个自动配置和启动的项来目解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

10 SpringCloud

Spring Cloud 是一系列框架的有序集合。它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、熔断器、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 并没有重复制造轮子，它只是将目前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。
Spring Cloud 专注于提供良好的开箱即用经验的典型用例和可扩展性机制覆盖。具有以下特性

分布式/版本化配置
服务注册和发现
路由
Service-to-service 调用
负载均衡
断路器
分布式消息传递

10.1 SpringCloud 和 SpringBoot 的关系

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于 Spring Boot 快速开发单
个微服务，Spring Cloud 是一个基于 Spring Boot 实现的云应用开发工具；Spring Boot 专
注于快速、方便集成的单个微服务个体，Spring Cloud 关注全局的服务治理框架；

沟通系Java工程师面试指导课程 -- samuel

Spring Boot 使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud 很大的一部分是基于 Spring Boot 来实现，可以不基于 Spring Boot 吗？不可以。

Spring Boot 可以离开 Spring Cloud 独立使用开发项目，但是 Spring Cloud 离不开 Spring Boot，属于依赖的关系。

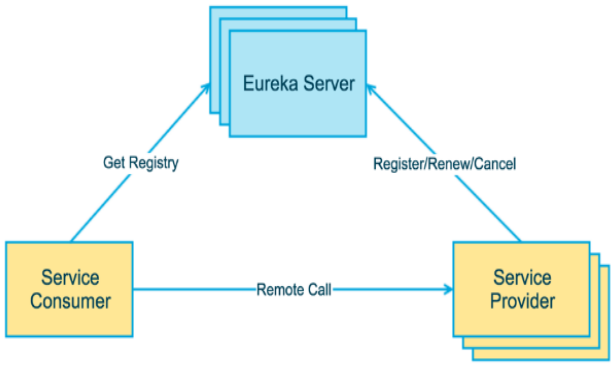
10.2 Eureka 介绍

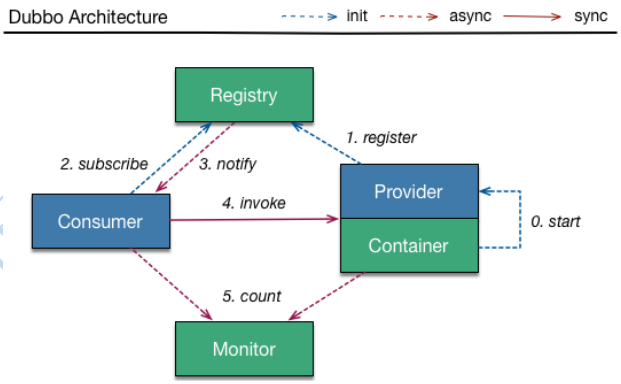
Eureka 是 Netflix 出品的用于实现服务注册和发现的工具。 Spring Cloud 集成了 Eureka，并提供了开箱即用的支持。其中， Eureka 又可细分为 Eureka Server 和 Eureka Client。

Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现(请对比 Zookeeper)。

Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。

而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。SpringCloud 的一些其他模块（比如 Zuul）就可以通过 Eureka Server 来发现系统中的其他微服务，并执行相关的逻辑。

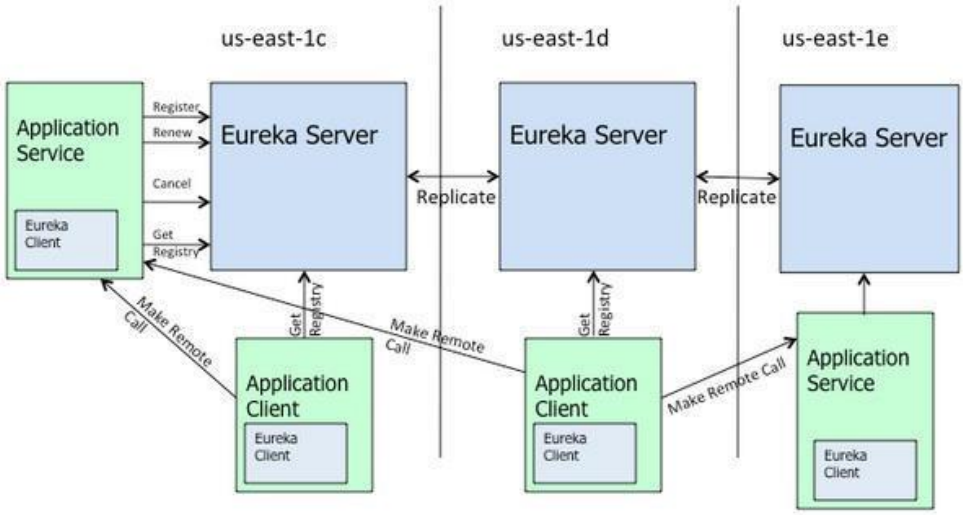




Eureka 包含两个组件：Eureka Server 和 Eureka Client。

- Eureka Server 提供服务注册服务,各个节点启动后会在 EurekaServer 中进行注册，这样 EurekaServer 中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到
- EurekaClient 是一个 Java 客户端，用于简化 Eureka Server 的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向 Eureka Server 发送心跳(默认周期为 30 秒)。如果 Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，EurekaServer 将会从服务注册表中把这个服务节点移除（默认 90 秒）

10.2.1 集群基本原理



上图是来自 eureka 的官方架构图，这是基于集群配置的 eureka；

- 处于不同节点的 eureka 通过 Replicate 进行数据同步
- Application Service 为服务提供者
- Application Client 为服务消费者
- Make Remote Call 完成一次服务调用

服务启动后向 Eureka 注册，Eureka Server 会将注册信息向其他 Eureka Server 进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

当服务注册中心 Eureka Server 检测到服务提供者因为宕机、网络原因不可用时，则在服务注册中心将服务置为 DOWN 状态，并把当前服务提供者状态向订阅者发布，订阅过的服务消费者更新本地缓存。

服务提供者在启动后，周期性(默认 30 秒)向 Eureka Server 发送心跳，以证明当前服务是可用状态。Eureka Server 在一定的时间（默认 90 秒）未收到客户端的心跳，则认为服务宕机，注销该实例。

10.2.2 Eureka 的自我保护机制

在默认配置中，Eureka Server 在默认 90s 没有得到客户端的心跳，则注销该实例，但是往往因为微服务跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，但是因为网络分区故障时，Eureka Server 注销服务实例则会让大部分微服务不可用，这很危险，因为服务明明没有问题。

为了解决这个问题，Eureka 有自我保护机制，通过在 Eureka Server 配置如下参数，可启动保护机制

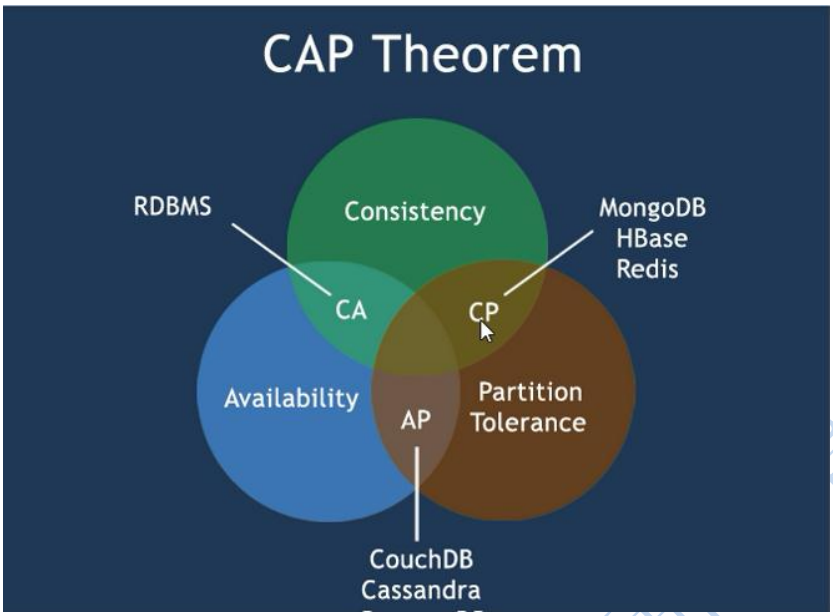
```
eureka.server.enable-self-preservation=true
```

它的原理是，当 Eureka Server 节点在短时间内丢失过多的客户端时（可能发送了网络故障），那么这个节点将进入自我保护模式，不再注销任何微服务，当网络故障回复后，该节点会自动退出自我保护模式。

自我保护模式的架构哲学是宁可放过一个，决不可错杀一千，好死不如赖活着

10.2.3 Eureka 和 Zookeeper 比较

CAP 理论的核心：一个分布式系统不可能同时很好地满足一致性、可用性和分区容错性这三个需求。因此，根据 CAP 原理将 NOSQL 数据分成了 CA 原则、CP 原则、AP 原则三大类：
CA：单点集群，满足一致性，可用性的系统，通常在可扩展性上不强
CP：满足一致性，分区容错性的系统，通常性能不够高
AP：满足可用用，分区容错性的系统，通常可能对一致性的要求低一些



CAP 原则又称 CAP 定理，指的是在一个分布式系统中，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可兼得。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在 A 和 C 之间进行权衡。在此 Zookeeper 保证的是 CP，而 Eureka 则是 AP。

10.2.3.1 Zookeeper 保证 CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接 down 掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是 zk 会出现这样一种情况，当 master 节点因为网络故障与其他节点失去联系时，剩余节点会重新进行 leader 选举。问题在于，选举 leader 的时间太长，30 ~ 120s, 且选举期间整个 zk 集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得 zk 集群失去 master 节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

10.2.3.2 Eureka 保证 AP

Eureka 看明白了这一点，因此在设计时就优先保证可用性。Eureka 各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而 Eureka 的客户端在向某个 Eureka 注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台 Eureka 还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka 还有一种自我保护机制，如果在 15 分钟内超过 85%的节点都没有正常的心跳，那么 Eureka 就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka 可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像 zookeeper 那样使整个注册服务瘫痪。

10.2.3.3 总结

Zookeeper 保证的是 CP, 而 Eureka 则是 AP

Eureka 作为单纯的服务注册中心来说要比 zookeeper 更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过 Eureka 目前 1.X 版本的实现是基于 servlet 的 Javaweb 应用，它的极限性能肯定会受到影响。期待正在开发之中的 2.X 版本能够从 servlet 中独立出来成为单独可部署执行的服务。

10.2.4 Eureka 的服务续约，失效剔除，和自我保护

失效剔除

有些时候， 我们的服务实例并不一定会正常下线， 可能由于内存溢出、 网络故障等原因使得服务不能正常工作，而服务注册中心并未收到“服务下线”的请求。为了从服务列表中将这些无法提供服务的实例剔除，Eureka Server 在启动的时候会创建一个定时任务，默认每隔一段时间（默认为 60 秒） 将当前清单中超时（默认为 90 秒）没有续约的服务剔除出去

服务续约

在注册完服务之后，服务提供者会维护一个心跳用来持续告诉 EurekaServer: "我还活着”， 以防止 Eureka Server 的“剔除任务 ” 将该服务实例从服务列表中排除出去，我们称该操作为服务续约(Renew)。

关千服务续约有两个重要属性，我们可以关注并根据需要来进行调整：

eureka.instance.lease-renewal-interval-in-seconds=30

eureka.instance.lease-expiration-duration-in-seconds=90

eureka.instance.lease-renewal-interval-in-seconds 参数用于定义服务续约任务的调用间隔时间，默认为 30 秒。

eureka.instance.lease-expiration-duration-in-seconds 参数用于定义服务失效的时间，默认为 90 秒

自我保护

服务注册到 EurekaServer 之后，会维护一个心跳连接，告诉 EurekaServer 自己还活着。EurekaServer 在运行期间，会统计心跳失败的比例在 15 分钟之内是否低于 85%, 如果出现低于的情况（在单机调试的时候很容易满足， 实际在生产环境上通常是由于网络不稳定导致）， Eureka Server 会将当前的实例注册信息保护起来， 让这些实例不会过期， 尽可能保护这些注册信息。

但是， 在这段保护期间内实例若出现问题， 那么客户端很容易拿到实际已经不存在的服务实例， 会出现调用失败的清况， 所以客户端必须要有容错机制， 比如可以使用请求重试、 断路器等机制。

由于本地调试很容易触发注册中心的保护机制， 这会使得注册中心维护的服务实例不那么准确。 所以， 我们在本地进行开发的时候， 可以使用 eureka.server.enable-self-preservaon = false 参数来关闭保护机制， 以确保注册中心可以将不可用的实例正确剔除。

10.3 Ribbon 和 Feign 的区别

Ribbon 和 Feign 都是用于调用其他服务的，不过方式不同。

- Ribbon 添加 maven 依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用 RestTemplate 调用远程服务对应的方法
- feign 添加 maven 依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用在接口上使用@FeignClient("指定服务名")
- 服务的指定位置不同，Ribbon 是在@RibbonClient 注解上声明，Feign 则是在定义抽象方法的接口中使用@FeignClient 声明。

10.4 springcloud 断路器的作用

当一个服务调用另一个服务由于网络原因或者自身原因出现问题时 调用者就会等待被调用者的响应 当更多的服务请求到这些资源时

导致更多的请求等待 这样就会发生连锁效应（雪崩效应） 断路器就是解决这一问题

断路器有完全打开状态

一定时间内 达到一定的次数无法调用 并且多次检测没有恢复的迹象 断路器完全打开，那么下次请求就不会请求到该服务

半开

短时间内 有恢复迹象 断路器会将部分请求发给该服务 当能正常调用时 断路器关闭

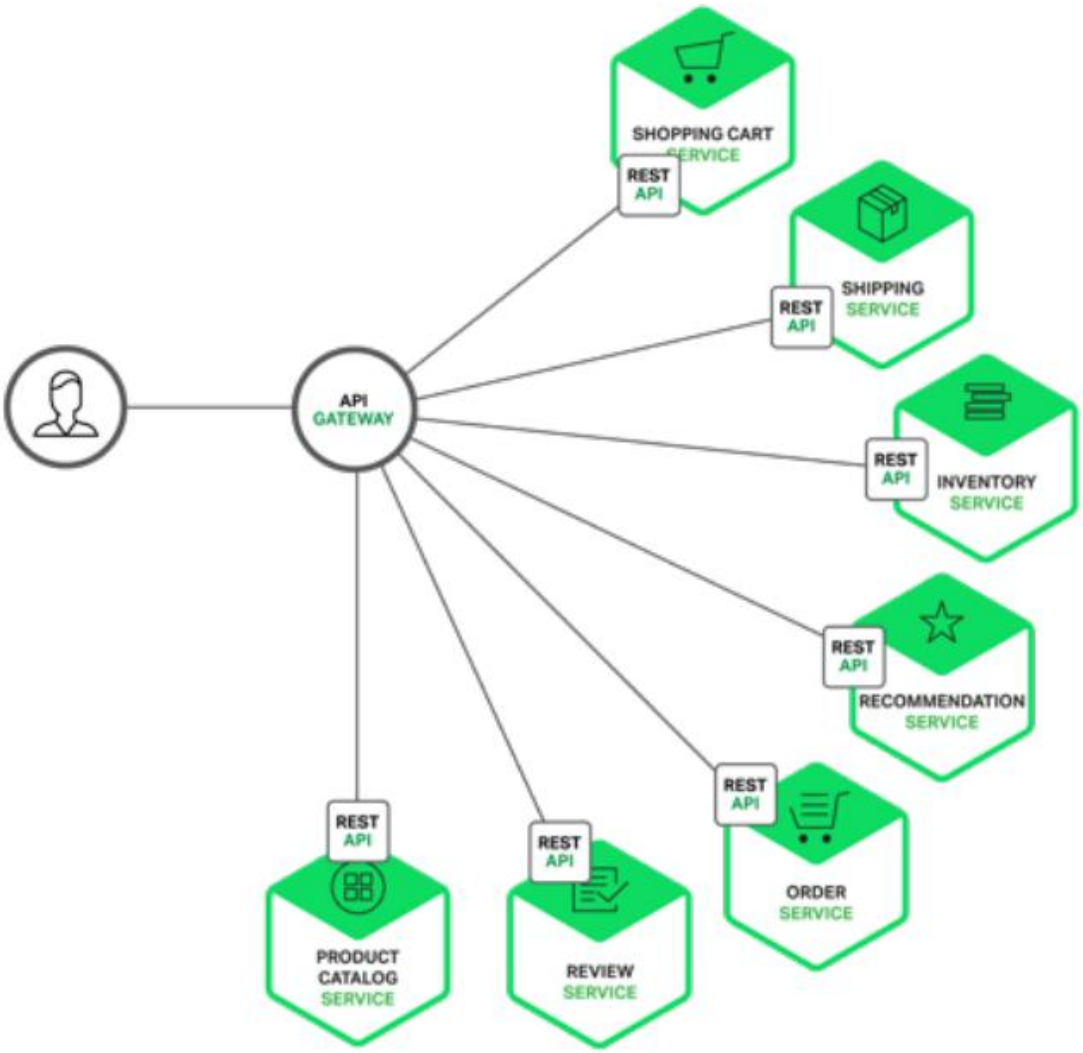
关闭

当服务一直处于正常状态 能正常调用 断路器关闭

10.5 API 网关

API Gateway 是一个服务器，也可以说是进入系统的唯一节点。这跟面向对象设计模式中的 Facade 模式【外观模式】很像。API Gateway 封装内部系统的架构，并且提供 API 给各个客户端。它还可能有 其他功能，如授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等。下图展示了一个适应当前架构的 API Gateway。

治愈系Java工程师面试指导课程——samuel



API Gateway 负责请求转发、合成和协议转换。所有来自客户端的请求都要先经过 API Gateway，然后路由这些请求到对应的微服务。

API Gateway 将经常通过调用多个微服务来处理一个请求以及聚合多个服务的结果。它可以在 web 协议与内部使用的非 Web 友好型协议间进行转换，如 HTTP 协议、WebSocket 协议。

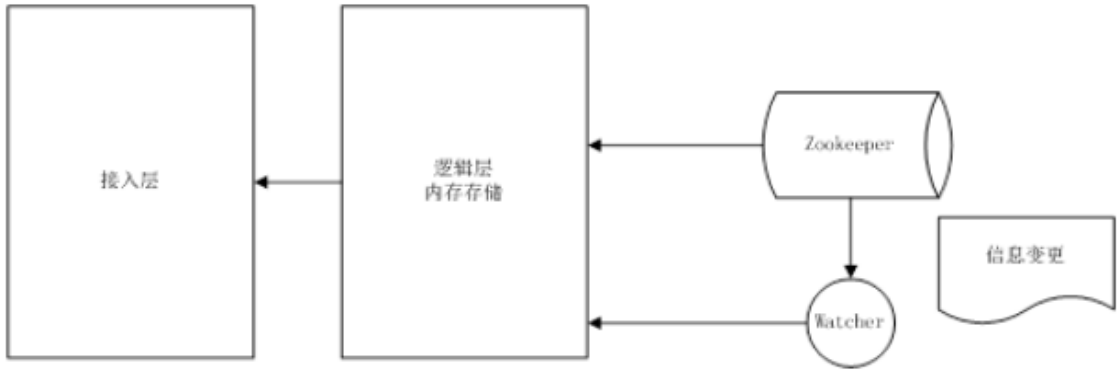
- 请求转发 ：服务转发主要是对客户端的请求安装微服务的负载转发到不同的服务上
- 响应合并：把业务上需要调用多个服务接口才能完成的工作合并成一次调用对外统一提供服务。
- 协议转换：重点是支持 SOAP，JMS，Rest 间的协议转换。
- 数据转换：重点是支持 XML 和 Json 之间的报文格式转换能力（可选）
- 安全认证：
  1. 基于 Token 的客户端访问控制和安全策略
  2. 传输数据和报文加密，到服务端解密，需要在客户端有独立的 SDK 代理包
  3. 基于 Https 的传输加密，客户端和服务端数字证书支持
  4. 基于 OAuth2.0 的服务安全认证(授权码，客户端，密码模式等）

10.6 配置中心

配置中心一般用作系统的参数配置，它需要满足如下几个要求：高效获取、实时感知、分布式访问。

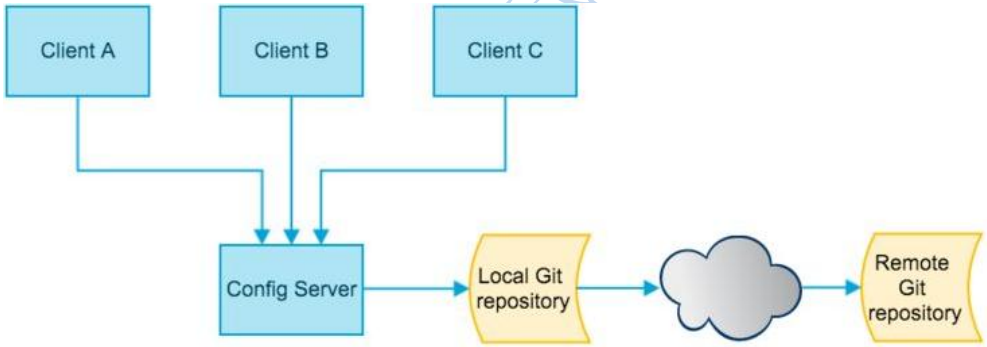
10.6.1 zookeeper 配置中心

实现的架构图如下所示，采取数据加载到内存方式解决高效获取的问题，借助 zookeeper 的节点 监听机制来实现实时感知。



10.6.2 SpringCloud Config

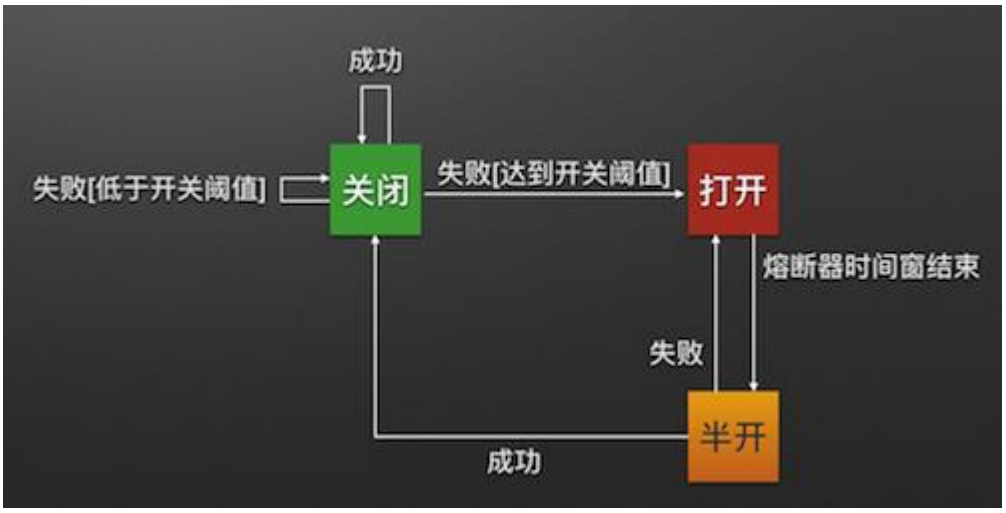
SpringCloud Config 为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。



10.7 服务熔断（Hystrix）

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个 系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不 可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到 许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序 不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费 CPU 时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经 修正，应用程序会再次尝试调用操作。



10.8 Hystrix 断路器机制

断路器很好理解，当 Hystrix Command 请求后端服务失败数量超过一定比例(默认 50%)，断路器会 切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态 一段时间后(默认 5 秒)，自动切换到半开路状态(HALF-OPEN). 这时会判断下一次请求的返回情况，如果请求成功，断路器切回闭路状态(CLOSED)，否则重新切换到开路状态(OPEN). Hystrix 的断路器 就像我们家庭电路中的保险丝，一旦后端服务不可用，断路器会直接切断请求链，避免发送大量无效 请求影响系统吞吐量，并且断路器有自我检测并恢复的能力。

治愈系Java工程师面试精品课程——samuel