1 深入理解abstract class 和 interface

1.1 从语法定义层面看

在面向对象领域，抽象类主要用来进行类型隐藏。我们可以构造出一个固定的一组行为的抽象描述，但是这组行为却能够有任意个可能的具体实现方式。

使用 abstract class 的方式定义 Demo 抽象类的方式如下：

```
abstract class Demo {
    abstract void method1();
    abstract void method2();
   ...
}
```

使用 interface 的方式定义 Demo 抽象类的方式如下：

```
interface Demo {
    void method1();
    void method2();
   ...
}
```

在abstract class 方式中，抽象类Demo可以有自己的数据成员，也可以有非abstract的成员方法，而interface方式的实现中，Demo只能够有静态的不能被修改的数据成员，也就是必须是static final的，不过interface中一般不定义数据成员，所有的成员方法都是abstract的，某种意义上说，interface是一种特殊的abstract class。

1.2 从编程层面看

首先，抽象类在java语言中表示的是一继承关系，一个类只能使用一次继承关系。但是一个类却可以实现多个interface。

其次，在抽象类定义中可以赋予方法的默认行为，但是在interface定义中方法不能有默认行为

1.3 从设计理念层面看

上面主要从语法定义和编程的角度论述了 abstract class 和 interface 的区别，这些层面的区别是比较低层次的、非本质的。本小节将从另一个层面：abstract class 和 interface 所反映出的设计理念，来分析一下二者的区别。作者认为，从这个层面进行分析才能理解二者概念的本质所在。

前面已经提到过，abstarct class 在 Java 语言中体现了一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在”is a”关系，即父类和派生类在概念本质上应该是相同的（参考文献〔3〕中有关于”is a”关系的大篇幅深入的论述，有兴趣的读者可以参考）。对于 interface 来说则不然，并不要求 interface 的实现者和 interface 定义在概念本质上是一致的，仅仅是实现了 interface 定义的契约而已。为了使论述便于理解，下面将通过一个简单的实例进行说明。

考虑这样一个例子，假设在我们的问题领域中有一个关于Door的抽象概念，该 Door 具有执行两个动作 open 和 close，此时我们可以通过 abstract class 或者 interface 来定义一个表示该抽象概念的类型，定义方式分别如下所示：

使用 abstract class 方式定义 Door：

```
abstract class Door {
        abstract void open();
        abstract void close();
}
```

使用 interface 方式定义 Door：

```
interface Door {
        void open();
    void close();
}
```

其他具体的 Door 类型可以 extends 使用 abstract class 方式定义的 Door 或者 implements 使用 interface 方式定义的 Door。看起来好像使用 abstract class 和 interface 没有大的区别。

如果现在要求 Door 还要具有报警的功能。我们该如何设计针对该例子的类结构呢（在本例中，主要是为了展示 abstract class 和 interface 反映在设计理念上的区别，其他方面无关的问题都做了简化或者忽略）？ 下面将罗列出可能的解决方案，并从设计理念层面对这些不同的方案进行分析。

解决方案一：

简单的在 Door 的定义中增加一个 alarm 方法，如下：

```
abstract class Door {
        abstract void open();
        abstract void close();
        abstract void alarm();
}
```

或者

```
interface Door {
        void open();
    void close();
    void alarm();
}
```

那么具有报警功能的 AlarmDoor 的定义方式如下：

```
class AlarmDoor extends Door {
        void open() {... }
            void close() {... }
        void alarm() {... }
}
```

或者

```
class AlarmDoor implements Door  {
    void open() {... }
            void close() {... }
        void alarm() {... }
}
```

这种方法违反了面向对象设计中的一个核心原则 ISP（Interface Segregation Priciple），在 Door 的定义中把 Door 概念本身固有的行为方法和另外一个概念"报警器"的行为方法混在了一起。这样引起的一个问题是那些仅仅依赖于 Door 这个概念的模块会因为"报警器"这个概念的改变（比如：修改 alarm 方法的参数）而改变，反之依然。

解决方案二：

既然 open、close 和 alarm 属于两个不同的概念，根据 ISP 原则应该把它们分别定义在代表这两个概念的抽象类中。定义方式有：这两个概念都使用 abstract class 方式定义；两个概念都使用 interface 方式定义；一个概念使用 abstract class 方式定义，另一个概念使用 interface 方式定义。

显然，由于 Java 语言不支持多重继承，所以两个概念都使用 abstract class 方式定义是不可行的。后面两种方式都是可行的，但是对于它们的选择却反映出对于问题领域中的概念本质的理解、对于设计意图的反映是否正确、合理。我们一一来分析、说明。

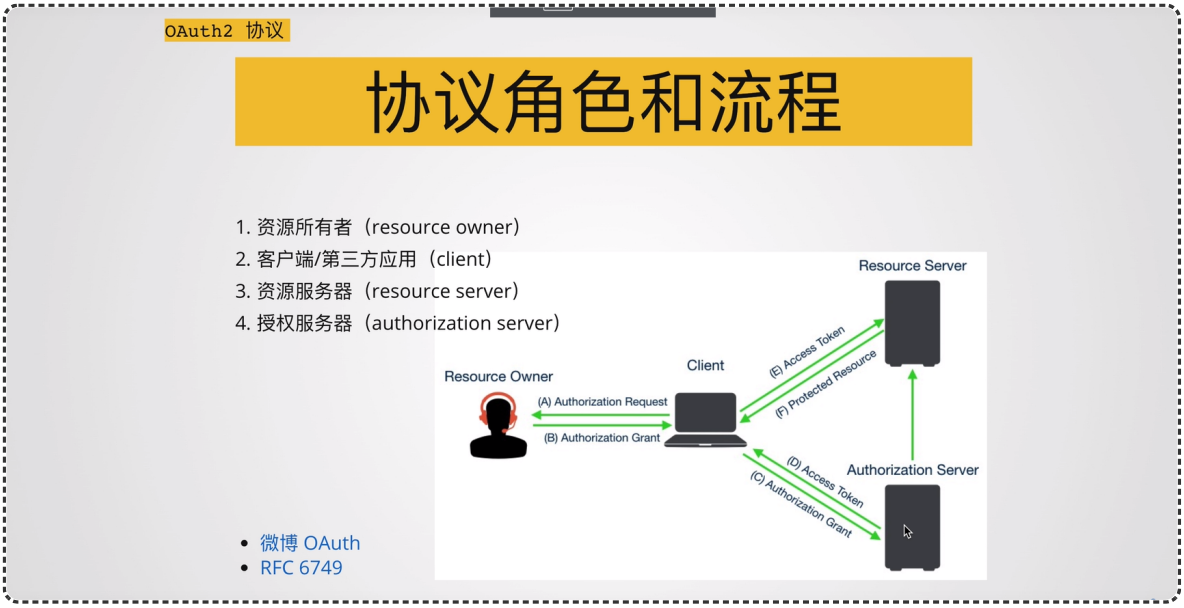
如果两个概念都使用 interface 方式来定义，那么就反映出两个问题：1、我们可能没有理解清楚问题领域，AlarmDoor 在概念本质上到底是Door还是报警器？2、如果我们对于问题领域的理解没有问题，比如：我们通过对于问题领域的分析发现 AlarmDoor 在概念本质上和 Door 是一致的，那么我们在实现时就没有能够正确的揭示我们的设计意图，因为在这两个概念的定义上（均使用 interface 方式定义）反映不出上述含义。

如果我们对于问题领域的理解是：AlarmDoor 在概念本质上是 Door，同时它有具有报警的功能。我们该如何来设计、实现来明确的反映出我们的意思呢？前面已经说过，abstract class 在 Java 语言中表示一种继承关系，而继承关系在本质上是"is a"关系。所以对于 Door 这个概念，我们应该使用 abstarct class 方式来定义。另外，AlarmDoor 又具有报警功能，说明它又能够完成报警概念中定义的行为，所以报警概念可以通过 interface 方式定义。如下所示：

```
abstract class Door {
        abstract void open();
        abstract void close();
    }
interface Alarm {
    void alarm();
}
class AlarmDoor extends Door implements Alarm {
    void open() {... }
    void close() {... }
       void alarm() {... }
}
```

这种实现方式基本上能够明确的反映出我们对于问题领域的理解，正确的揭示我们的设计意图。其实 abstract class 表示的是"is a"关系，interface 表示的是"like a"关系，大家在选择时可以作为一个依据，当然这是建立在对问题领域的理解上的，比如：如果我们认为AlarmDoor在概念本质上是报警器，同时又具有 Door 的功能，那么上述的定义方式就要反过来了。

2 OAuth2



2.1 一、写在前面

在收集资料时，我查询和学习了许多介绍OAuth的文章，这些文章有好有坏，但大多是从个例出发。因此我想从官方文档出发，结合在stackoverflow上的一些讨论，一并整理一下。整理的内容分为OAuth1.0a和OAuth2两部分。

> OAuth 1.0a：One Leg ->Two Leg -> Three Legged
> OAuth 2:Two Leg ->Three Legged (附：Refresh Token的方式)

这两种模式都是按箭头从左往右安全性递增，其实现也会相对复杂。关于官方的这种leg（腿？）的说法，在中文翻译中比较少有文章提及。下面分别来介绍OAuth的这5种授权流程。

2.2 二、OAuth1.0a

2.1 OAuth 1.0a (One Leg)

1. 应用给服务器发送一个签名请求，附带以下参数：

    - oauth_token Empty String
    - oauth_consumer_key
    - oauth_timestamp
    - oauth_nonce
    - oauth_signature
    - oauth_signature_method
    - oauth_version Optional
2. 服务验证并授予对资源的访问
3. 应用程序利用请求的资源

2.2 OAuth 1.0a (Two Legs)

1. 应用发送一个签名请求，以获取 Request Token：
    - oauth_consumer_key
    - oauth_timestamp
    - oauth_nonce
    - oauth_signature
    - oauth_signature_method
    - oauth_version Optional
2. 服务器返回Request Token：
    - oauth_token
    - oauth_token_secret
    - Additional Parameters / Arguments
3. 发送签名请求，用Request Token换取Access Token
    - oauth_token Request Token
    - oauth_consumer_key
    - oauth_nonce
    - oauth_signature
    - oauth_signature_method
    - oauth_version
4. 服务器返回Access Token和Token Secret
5. 应用通过Access Token和Token Secret利用请求的资源

2.3 OAuth 1.0a (Three Legged)

1. 应用发送一个签名请求，以获取 Request Token：
    - oauth_consumer_key
    - oauth_timestamp
    - oauth_nonce
    - oauth_signature
    - oauth_signature_method
    - oauth_version Optional
2. 服务器返回Request Token：
    - oauth_token
    - oauth_token_secret
    - oauth_callback_confirmed
    - … Additional Parameters / Arguments
3. 发送给用户授权的URL
    - oauth_token
4. 提示用户进行授权
5. 用户进行授权

6. 授权结束后返回应用，附带上：
    - oauth_token
    - oauth_verifier
7. 发送签名请求，用Request Token换取Access Token
    - oauth_token Request Token
    - oauth_consumer_key
    - oauth_nonce
    - oauth_signature
    - oauth_signature_method
    - oauth_version
    - oauth_verifier
8. 服务器返回Access Token和Token Secret
9. 应用通过Access Token和Token Secret利用请求的资源

2.3 三、OAuth2

3.1 OAuth 2 (Two Legged)

3.1.1 客户端凭据方式

1. 应用发送请求到服务器：
    - grant_type = client_credentials
      如果没有使用Authorization（Authorization: Basic Base64(client_id:client_secret)） 的header，必须附带参数为：
    - client_id
    - client_secret
2. 服务器以Access Token回应
    - access_token
    - expires_in
    - token_type

3.1.2 隐式授予方式

1. 应用发送请求到服务器：
    - response_type = token
    - redirect_uri This is a server-side Redirection URI hosted by the provider or yourself.
    - scope
    - state Optional
    - client_id
2. 用户可根据需要授权。
    - username
    - password
3. 服务器将响应包含access_token在内的redirect_uri
4. 应用程序跳转至redirect_uri
5. redirect_uri将响应一段脚本或HTML片段。响应的脚本或HTML片段包含参数access_token，还有您可能需要的任何其他参数。

3.1.3 资源所有者密码方式

1. 应用向资源所有者请求凭证
    - username

- password

2. 应用使用凭证，向服务器发送请求
    - grant_type = password
    - username
    - password
      url看起来会像这样：
      grant_type=password&username=my_username&password=my_password
      如果你没有使用Authorization的header，必须附带上参数：
    - client_id
    - client_secret
      url看起来会像是：
      grant_type=password&username=my_username&password=my_password&client_id=random_string&client_secret=random_secret
3. 服务器返回Access Toke
    - access_token
    - expires_in
    - token_type

3.2 OAuth 2 (Three Legged)

1. 应用重定向用户到授权服务：
    - client_id
    - redirect_uri
    - response_type
    - state Optional; Unique identifier to protect against CSRF
    - scope Optional; what data your application can access.
      url看起来会像是：
      oauth_service/login/oauth/authorize?client_id=3MVG9lKcPoNINVB&redirect_uri=http://localhost/oauth/code_callback&scope=user
2. 用户登录服务器并确认授权给应用
3. 服务器重定向用户到redirect_url ，附带参数：
    - code
    - state
4. 应用拿到code，并换取Access Token
    - client_id
    - client_secret
    - code
    - redirect_uri Optional;
    - grant_type = “authorization_code”
5. 如果的client_id和client_secret是有效的，服务器将调用一个回调redirect_url，包含ACCESS_TOKEN
    - access_token
    - expires_in
    - refresh_token
6. 应用保存ACCESS_TOKEN，在随后的请求中使用。通常这个值被存储在session或或cookie，需要时作为授权请求的参数。

3.3 OAuth 2 (Refresh Token 刷新token)

在OAuth2中，Token会有过期时间，我们必须去refresh_token，使用其他一些先前获得的参数，生成一个新的token。这是一个容易得多的流程。

1. 创建刷新令牌请求
    - grant_type = “refresh_token”
    - scope Optional; Cannot have any new scopes not previously defined.
    - refresh_token
    - client_id
    - client_secret
2. 服务验证和响应以下参数：
    - access_token
    - issued_at

2.4 四、stackoverflow上的一些问答

Q：OpenID和OAuth的区别是什么?
A：OpenID是有关身份验证（即证明你是谁），OAuth有关授权（即授予访问权限），推荐博文：从用户的角度来看OpenID和OAuth

Q：OAuth2与OAuth1不同的地方是？有人可以简单的解释的OAuth2和OAuth1之间的区别吗？OAuth1现在已经过时，应实施的OAuth2？我没有看到许多实现的OAuth2，大多数仍在使用OAuth，这让我怀疑的OAuth2的准备使用。是吗?
A：OAuth2能更好地支持不是基于浏览器的应用。对于不是基于浏览器的应用程序，这是对OAuth的主要挑战。例如，在OAuth1.0，桌面应用或手机应用必须引导用户打开浏览器所需的服务，与服务进行身份验证，并复制令牌从服务返回给应用程序。这里的主要批评是针对用户体验。使用OAuth2.0，可以用新的方式为用户的应用程序获得授权。
OAuth2.0不再需要客户端应用程序拥有密钥。这让人回想起老的Twitter认证的API，它并不需要应用得到HMAC哈希令牌和请求字符串。使用OAuth2.0，应用程序可以通过HTTPS获得令牌。
OAuth2.0的签名流程简单得多。没有更多的特殊解析，排序，或编码。
OAuth2.0的访问令牌是“短命”的。通常情况下，OAuth1.0的访问令牌可以存储一年或一年以上（Twitter从来没有让他们到期）。 OAuth的2.0有刷新令牌的概念。虽然我不能完全肯定这是什么意思，我的猜测是，您的访问令牌可以是短暂存储的（即基于会话），而你可以刷新令牌。你使用刷新令牌获取新的访问令牌，而不是让用户重新授权您的应用程序。
最后，OAuth2.0使得负责处理的OAuth请求的服务器和处理用户的授权服务器之间的角色有一个干净的分离。更多信息，在上述的文章中详述。

Q：OAuth2服务器群怎么使用state来防范CSRF?
A：state只是一个随机的字符串，可以做这样的事情：$state = md5(uniqid(rand(), TRUE));在session中记录satate，以便稍后你能做验证。一些额外的资料：OAuth2威胁文件模型，特别CSRF保护

3 Spring MVC

基于Servlet API构建的原始Web框架，并且围绕着前端控制器模式进行设计

3.1 DispatcherServlet

SpringMVC 围绕着前端控制器模式进行设计，`DispatcherServlet`提供了用于请求处理的共享算法，并且和其他Servlet一样，需要根据Servlet规范通过使用Java配置或者在web.xml中进行声明和map。反过来，`DispatcherServlet`使用Spring配置发现请求Map，视图解析，异常处理等所需的委托组件

以下 Java 配置示例注册并初始化`DispatcherServlet`，它由 Servlet 容器自动检测(请参见Servlet Config)：

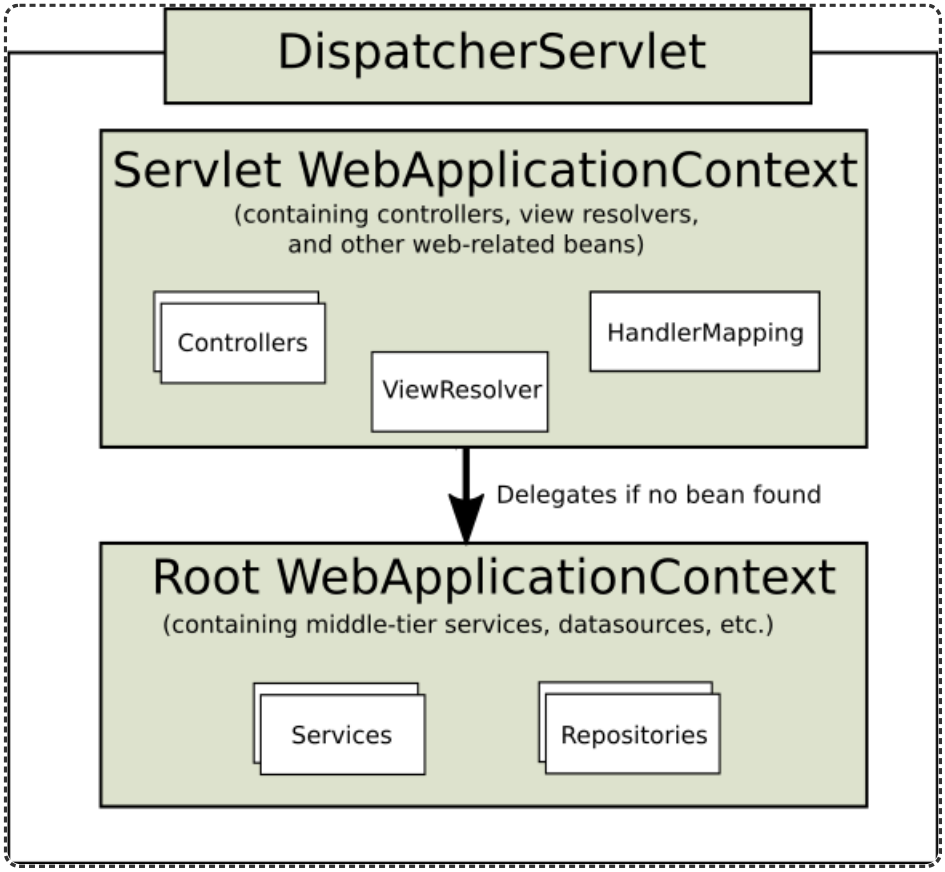
```java
public class MyWebApplicationInitializer implements
WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletCxt) {

        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext ac = new
AnnotationConfigWebApplicationContext();
        ac.register(AppConfig.class);
        ac.refresh();

        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(ac);
        ServletRegistration.Dynamic registration =
servletCxt.addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

3.1.1 上下文层次

`DispatcherServlet` 期望其自己的配置为 `WebApplicationContext` (纯 `ApplicationContext` 的 extensions)。 `WebApplicationContext` 具有到 `ServletContext` 和与其关联的 `Servlet` 的链接。它还绑定到 `ServletContext` ，以便应用程序可以在 `RequestContextUtils` 上使用静态方法来查找 `WebApplicationContext` (如果需要访问它们)。

对于许多应用程序来说，只有一个 `WebApplicationContext` 很简单并且足够。也可能具有上下文层次结构，其中一个根 `WebApplicationContext` 跨多个 `DispatcherServlet` (或其他 `Servlet`)实例共享，每个实例都有其自己的子 `WebApplicationContext` 配置。有关上下文层次结构功能的更多信息，请参见ApplicationContext 的其他功能。

根 `WebApplicationContext` 通常包含需要在多个 `Servlet` 实例之间共享的基础结构 Bean，例如数据存储库和业务服务。这些 Bean 是有效继承的，可以在 Servlet 特定子 `WebApplicationContext` 中重写(即重新声明)，该子 `WebApplicationContext` 通常包含给定 `Servlet` 本地的 Bean。下图显示了这种关系：



以下示例配置`WebApplicationContext`层次结构：

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { App1Config.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/app1/*" };
    }
}
```

Tip：

如果不需要应用程序上下文层次结构，则应用程序可以通过`getServletConfigClasses()`的`getRootConfigClasses()`和`null`返回所有配置。

以下示例显示了`web.xml`等效项：

```
<web-app>

    <listener>
```

```
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app1</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/app1-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>app1</servlet-name>
        <url-pattern>/app1/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Tip：

如果不需要应用程序上下文层次结构，则应用程序可以仅配置“根”上下文，并将 `contextConfigLocation` Servlet 参数保留为空。

3.1.2 特殊 bean 类

与 Spring WebFlux 中的相同

`DispatcherServlet` 委托特殊 bean 处理请求并呈现适当的响应。所谓“特殊 bean”，是指实现 WebFlux 框架 Contract 的 SpringManagement 的 `Object` 实例。这些通常带有内置 Contract，但是您可以自定义它们的属性并扩展或替换它们。

下表列出了 `DispatcherHandler` 检测到的特殊 bean：

Bean type	Explanation
HandlerMapping处理器映射器	将请求与interceptors列表一起映射到处理程序，以进行预处理和后期处理。映射基于某些条件，具体取决于`HandlerMapping`实现。`HandlerMapping`的两个主要实现是`RequestMappingHandlerMapping`(支持`@RequestMapping`带注解的方法)和`SimpleUrlHandlerMapping`(将 URI 路径模式的显式注册维护到处理程序)。
HandlerAdapter处理器适配器	帮助DispatcherServlet调用映射到请求的处理程序，而不管处理程序实际是如何调用的。例如，调用带注释的控制器需要解析注释。HandlerAdapter的主要目的是保护DispatcherServlet不受这些细节的影响。
HandlerExceptionResolver	解决异常的策略，可能将它们映射到处理程序、HTML错误视图或其他目标。
`ViewResolver`视图解析器	解析从处理程序返回到实际`View`的基于逻辑`String`的视图名称，使用该视图名称呈现给响应的实际视图。。参见View Resolution和View Technologies。
`LocaleResolver`, LocaleContextResolver	解析 Client 端正在使用的`Locale`以及可能的时区，以便能够提供国际化的视图。参见Locale。
ThemeResolve	解决 Web 应用程序可以使用的主题，例如提供个性化的布局。参见Themes。
MultipartResolver	用于借助一些 Multipart 解析库来分析 Multipart 请求(例如，浏览器表单文件上载)的抽象。参见Multipart Resolver。
FlashMapManager	存储和检索“input”和“output”管理FlashMap，它们可用于将属性从一个请求传递到另一个请求，通常是通过重定向。参见Flash Attributes。

3.1.3 Web MVC 配置

应用程序可以声明处理请求所必需的特殊 bean 类中列出的基础结构 bean。`DispatcherServlet`为每个特殊 bean 检查`WebApplicationContext`。如果没有匹配的 Bean 类型，它将使用DispatcherServlet.properties中列出的默认类型。

在大多数情况下，MVC Config是最佳起点。它使用 Java 或 XML 声明所需的 bean，并提供更高级别的配置回调 API 对其进行自定义。

Note：

Spring Boot 依靠 MVC Java 配置来配置 Spring MVC，并提供许多额外的方便选项。

3.1.4 Servlet 配置

在 Servlet 3.0 环境中，您可以选择以编程方式配置 Servlet 容器，以替代方式或与`web.xml`文件结合使用。下面的示例注册一个`DispatcherServlet`：

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements
WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new
XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-
config.xml");

        ServletRegistration.Dynamic registration =
container.addServlet("dispatcher", new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

`WebApplicationInitializer`是 Spring MVC 提供的接口，可确保检测到您的实现并将其自动用于初始化任何 Servlet 3 容器。名为`AbstractDispatcherServletInitializer`的`WebApplicationInitializer`的抽象 Base Class 实现通过覆盖方法来指定 servletMap 和`DispatcherServlet`配置的位置，从而使`DispatcherServlet`的注册更加容易。

对于使用基于 Java 的 Spring 配置的应用程序，建议这样做，如以下示例所示：

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

如果使用基于 XML 的 Spring 配置，则应直接从`AbstractDispatcherServletInitializer`扩展，如以下示例所示：

```
public class MyWebAppInitializer extends
AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

`AbstractDispatcherServletInitializer`还提供了一种方便的方法来添加`Filter`实例，并将它们自动 Map 到`DispatcherServlet`，如以下示例所示：

```
public class MyWebAppInitializer extends
AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {
            new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };
    }
}
```

每个过滤器都会根据其具体类型添加一个默认名称，并自动 Map 到`DispatcherServlet`。

`AbstractDispatcherServletInitializer`的受`isAsyncSupported`保护的方法提供了一个位置，以对`DispatcherServlet`及其 Map 的所有过滤器启用异步支持。默认情况下，此标志设置为`true`。

最后，如果您需要进一步自定义`DispatcherServlet`本身，则可以覆盖`createDispatcherServlet`方法。

3.1.5 Processing

`DispatcherServlet`处理请求的方式如下：

- 搜索`WebApplicationContext`并将其绑定在请求中，作为控制器和流程中其他元素可以使用的属性。默认情况下，它是在`DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`键下绑定的。
- locale resolver语言环境解析器绑定到请求，以使流程中的元素解析在处理请求(呈现视图，准备数据等)时要使用的语言环境。如果不需要语言环境解析，则不需要语言环境解析器。
- theme resolver主题解析器绑定到请求，以使诸如视图之类的元素确定要使用的主题。如果不使用主题，则可以忽略它。

- 如果指定 Multipart file resolver 文件解析器，则将检查请求中是否有 Multipart。如果找到 Multipart，则将请求包装在`MultipartHttpServletRequest`中，以供流程中的其他元素进一步处理。有关 Multipart 处理的更多信息，请参见Multipart Resolver。
- 搜索适当的处理程序。如果找到处理程序，则执行与处理程序(预处理器，后处理器和控制器)关联的执行链，以准备模型或渲染。或者，对于带注解 的控制器，可以呈现响应(在`HandlerAdapter`内)，而不返回视图。
- 如果返回模型，则呈现视图。如果没有返回任何模型(可能是由于预处理器或后处理器拦截了该请求，可能出于安全原因)，则不会呈现任何视图，因为该请求可能已经被满足。

`WebApplicationContext`中声明的`HandlerExceptionResolver` bean 用于解决在请求处理期间引发的异常。这些异常解析器允许定制逻辑以解决异常。有关更多详细信息，请参见Exceptions。

Spring `DispatcherServlet`还支持 Servlet API 指定的`last-modification-date`的返回。确定特定请求的最后修改日期的过程很简单：`DispatcherServlet`查找适当的处理程序 Map 并测试找到的处理程序是否实现`LastModified`接口。如果是这样，则`LastModified`接口的`long getLastModified(request)`方法的值返回给 Client 端。

您可以通过向`web.xml`文件中的 Servlet 声明中添加 Servlet 初始化参数(`init-param`元素)来自定义`DispatcherServlet`实例。下表列出了受支持的参数：

表 1. DispatcherServlet 初始化参数

Parameter	Explanation
`contextClass`	实现`ConfigurableWebApplicationContext`的类，将由此 Servlet 实例化并在本地配置。默认情况下，使用`XmlWebApplicationContext`。
`contextConfigLocation`	传递给上下文实例的字符串(由`contextClass`指定)，以指示可以在哪里找到上下文。该字符串可能包含多个字符串(使用逗号作为分隔符)以支持多个上下文。对于具有两次定义的 bean 的多个上下文位置，以最新位置为准。
`namespace`	`WebApplicationContext`的命名空间。默认为`[servlet-name]-servlet`。
`throwExceptionIfNoHandlerFound`	在找不到请求处理程序时是否抛出`NoHandlerFoundException`。然后可以使用`HandlerExceptionResolver`捕获异常(例如，通过使用`@ExceptionHandler`控制器方法)，然后将其作为其他任何异常进行处理。

默认情况下，它设置为`false`，在这种情况下`DispatcherServlet`将响应状态设置为404(NOT_FOUND)，而不会引发异常。
请注意，如果还配置了默认 servlet 处理，则始终将未解决的请求转发到默认 servlet，并且永远不会引发 404.

3.1.6 Interception

所有`HandlerMapping`实现都支持处理程序拦截器，这些拦截器在您要将特定功能应用于某些请求时非常有用-例如检查主体。拦截器必须使用三种方法从`org.springframework.web.servlet`包中实现`HandlerInterceptor`，这三种方法应提供足够的灵活性以执行所有类型的预处理和后处理：

- `preHandle(..)`：在执行实际处理程序之前
- `postHandle(..)`：执行处理程序后
- `afterCompletion(..)`：完成完整的请求后

`preHandle(..)`方法返回布尔值。您可以使用此方法来中断或 continue 执行链的处理。当此方法返回`true`时，处理程序执行链 continue。当它返回 false 时，`DispatcherServlet`假定拦截器本身已经处理了请求(例如，提供了适当的视图)，并且不会 continue 执行执行链中的其他拦截器和实际处理程序。

有关如何配置拦截器的示例，请参见 MVC 配置部分中的Interceptors。您还可以通过使用各个`HandlerMapping`实现上的设置器直接注册它们。

请注意，`postHandle`在`@ResponseBody`和`ResponseEntity`方法中用处不大，因为在`HandlerAdapter`内和`postHandle`之前将响应写入和提交。这意味着对响应进行任何更改为时已晚，例如添加额外的 Headers。对于此类情况，您可以实现`ResponseBodyAdvice`并将其声明为Controller Advice bean 或直接在`RequestMappingHandlerAdapter`上对其进行配置。

3.1.7 Exceptions

如果在请求 Map 期间发生异常或从请求处理程序(例如`@Controller`)引发异常，则`DispatcherServlet`委托`HandlerExceptionResolver` bean 链来解决该异常并提供替代处理，通常是错误响应。

下表列出了可用的`HandlerExceptionResolver`实现：

表 2. HandlerExceptionResolver 实现

`HandlerExceptionResolver`	**Description**
`SimpleMappingExceptionResolver`	异常类名和错误视图名之间的映射。用于在浏览器应用程序中呈现错误页。
DefaultHandlerExceptionResolver	解决springmvc引发的异常，并将它们映射到HTTP状态代码。另请参阅备选ResponseEntityExceptionHandler和RESTAPI异常。
`ResponseStatusExceptionResolver`	使用@ResponseStatus注释解析异常，并根据注释中的值将它们映射到HTTP状态代码。
`ExceptionHandlerExceptionResolver`	通过调用@Controller或@ControllerAdvice类中的@ExceptionHandler方法来解决异常。请参见@ExceptionHandler方法。

3.1.7.1 Chain of Resolvers解析器链

您可以通过在 Spring 配置中声明多个`HandlerExceptionResolver` bean 并根据需要设置其`order`属性来形成异常解析器链。 order 属性越高，异常解析器的定位就越晚。

`HandlerExceptionResolver`的 Contract 规定它可以返回：

- `ModelAndView`指向错误视图。
- 如果在解析程序中处理了异常，则为空`ModelAndView`。
- `null`如果仍未解决异常，则供以后的解析器尝试，并且，如果异常仍在末尾，则允许其冒泡到 Servlet 容器。

MVC Config自动为默认的 Spring MVC 异常，`@ResponseStatus`带注解 的异常以及对`@ExceptionHandler`方法的支持声明内置解析器。您可以自定义该列表或替换它。

3.1.7.2 Container Error Page 容器错误页

如果任何`HandlerExceptionResolver`都无法解决异常，因此该异常可以 continue 传播，或者如果响应状态设置为错误状态(即 4xx，5xx)，则 Servlet 容器可以在 HTML 中呈现默认错误页面。要自定义容器的默认错误页面，可以在`web.xml`中声明一个错误页面 Map。以下示例显示了如何执行此操作：

```
<error-page>
    <location>/error</location>
</error-page>
```

给定前面的示例，当异常冒出气泡或响应具有错误状态时，Servlet 容器在容器内向配置的 URL(例如`/error`)进行 ERROR 调度。然后由`DispatcherServlet`处理，可能将其映射到`@Controller`，可以实现该错误以返回带有模型的错误视图名称或呈现 JSON 响应，如以下示例所示：

```
@RestController
public class ErrorController {

    @RequestMapping(path = "/error")
    public Map<String, Object> handle(HttpServletRequest request) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status",
request.getAttribute("javax.servlet.error.status_code"));
        map.put("reason",
request.getAttribute("javax.servlet.error.message"));
        return map;
    }
}
```

Tip：

Servlet API 没有提供在 Java 中创建错误页面映射的方法。但是，您可以同时使用`WebApplicationInitializer`和最小的`web.xml`。

3.1.8 View Resolution 视图分辨率

springmvc定义了ViewResolver和视图接口，这些接口允许您在浏览器中呈现模型，而无需将您绑定到特定的视图技术。ViewResolver提供视图名称和实际视图之间的映射。视图处理在移交给特定视图技术之前的数据准备。

下表提供了有关ViewResolver层次结构的详细信息：

表 3. ViewResolver 实现

ViewResolver	Description
`AbstractCachingViewResolver`	`AbstractCachingViewResolver` 缓存视图实例所解析的子类。缓存可以提高某些视图技术的性能。您可以通过将 `cache` 属性设置为 `false` 来关闭缓存。此外，如果必须在运行时刷新某个视图(例如，修改 FreeMarker 模板时)，则可以使用 `removeFromCache(String viewName, Locale loc)` 方法。
`XmlViewResolver`	`ViewResolver` 的实现，该实现接受用 XML 编写的配置文件，该配置文件的 DTD 与 Spring 的 XML bean 工厂相同。默认配置文件为 `/WEB-INF/views.xml` 。
`ResourceBundleViewResolver`	`ViewResolver` 的实现使用 `ResourceBundle` 中的 bean 定义(由包基本名称指定)。对于应该解析的每个视图，它将属性 `[viewname].(class)` 的值用作视图类，并将属性 `[viewname].url` 的值用作视图 URL。您可以在View Technologies的章节中找到示例。
`UrlBasedViewResolver`	`ViewResolver` 接口的简单实现会影响逻辑视图名称到 URL 的直接解析，而无需显式 Map 定义。如果您的逻辑名称以直接的方式与视图资源的名称匹配，而不需要任意 Map，则这是适当的。
`InternalResourceViewResolver`	方便的 `UrlBasedViewResolver` 子类，支持 `InternalResourceView` (实际上是 Servlet 和 JSP)以及 `JstlView` 和 `TilesView` 之类的子类。您可以使用 `setViewClass(..)` 为该解析器生成的所有视图指定视图类。有关详细信息，请参见UrlBasedViewResolver javadoc。
`FreeMarkerViewResolver`	`UrlBasedViewResolver` 的便捷子类，支持 `FreeMarkerView` 及其自定义子类。
`ContentNegotiatingViewResolver`	`ViewResolver` 接口的实现，该接口根据请求文件名或 `Accept` Headers 解析视图。参见Content Negotiation。

3.1.8.1 Handling

您可以通过声明多个解析器bean来链接视图解析器，如果需要，还可以通过设置order属性来指定顺序。请记住，order属性越高，视图解析器在链中的位置就越晚。

ViewResolver的协定指定它可以返回null以指示找不到视图。但是，对于JSP和InternalResourceViewResolver，确定JSP是否存在的唯一方法是通过RequestDispatcher执行调度。因此，必须始终将InternalResourceViewResolver配置为视图解析程序总体顺序中的最后一个。

配置视图解析就像在Spring配置中添加ViewResolver bean一样简单。mvcconfig为视图解析器和添加无逻辑视图控制器提供了一个专用的配置API，这对于没有控制器逻辑的HTML模板渲染非常有用。

3.1.8.2 Redirecting

视图名称中的特殊重定向：前缀允许执行重定向。UrlBasedViewResolver（及其子类）将其识别为需要重定向的指令。视图名称的其余部分是重定向URL。

最终效果与控制器返回 `RedirectView` 的效果相同，但是现在控制器本身可以根据逻辑视图名称进行操作。逻辑视图名称(例如 `redirect:/myapp/some/resource`)相对于当前 Servlet 上下文重定向，而名称(例如 `redirect:http://myhost.com/some/arbitrary/path`)重定向到绝对 URL。

请注意，如果控制器方法用 `@ResponseStatus` Comments，则注解 值优先于 `RedirectView` 设置的响应状态。

3.1.8.3 Forwarding

您还可以为视图名称使用特殊的 `forward:` 前缀，这些名称最终由 `UrlBasedViewResolver` 和子类解析。这将创建一个 `InternalResourceView` ，它执行 `RequestDispatcher.forward()` 。因此，此前缀对于 `InternalResourceViewResolver` 和 `InternalResourceView` (对于 JSP)没有用，但是如果您使用另一种视图技术，但仍然希望强制转发由 Servlet/JSP 引擎处理的资源，则该前缀很有用。请注意，您也可以改为链接多个视图解析器。

3.1.8.4 Content Negotiation

ContentNegotiatingViewResolver本身不会解析视图，而是委派给其他视图解析器，并选择类似于 Client 端请求的表示形式的视图。可以从 `Accept` Headers 或查询参数(例如 `"/path?format=pdf"`)确定表示形式。

`ContentNegotiatingViewResolver` 通过将请求媒体类型与与每个 `ViewResolvers` 关联的 `View` 支持的媒体类型(也称为 `Content-Type`)进行比较，从而选择合适的 `View` 来处理请求。列表中具有兼容 `Content-Type` 的第一个 `View` 将表示形式返回给 Client 端。如果 `ViewResolver` 链无法提供兼容的视图，请查阅通过 `DefaultViews` 属性指定的视图列表。后一个选项适用于单例 `Views` ，该单例 `Views` 可以呈现当前资源的适当表示形式，而与逻辑视图名称无关。 `Accept` Headers 可以包含通配符(例如 `text/*`)，在这种情况下， `Content-Type` 为 `text/xml` 的 `View` 是兼容的匹配。

3.1.9 Locale

正如 Spring Web MVC 框架所做的那样，Spring 体系结构的大多数部分都支持国际化。 `DispatcherServlet` 使您可以使用 Client 端的语言环境自动解析邮件。这是通过 `LocaleResolver` 个对象完成的。

收到请求时， `DispatcherServlet` 查找语言环境解析器，如果找到一个，它将尝试使用它来设置语言环境。通过使用 `RequestContext.getLocale()` 方法，您始终可以检索由语言环境解析器解析的语言环境。

除了自动的语言环境解析之外，您还可以在处理程序 Map 上附加一个拦截器(有关处理程序 Map 拦截器的更多信息，请参见Interception)以在特定情况下(例如，基于请求中的参数)更改语言环境。

语言环境解析器和拦截器在 `org.springframework.web.servlet.i18n` 包中定义，并以常规方式在您的应用程序上下文中配置。 Spring 包含以下选择的语言环境解析器。

- Time Zone
- Header Resolver
- Cookie Resolver
- Session Resolver
- Locale Interceptor

3.1.9.1 Time Zone

除了获取 Client 的语言环境外，了解其时区通常也很有用。 `LocaleContextResolver` 界面提供了对 `LocaleResolver` 的扩展，该扩展使解析器可以提供更丰富的 `LocaleContext` ，其中可能包含时区信息。

如果可用，可以使用`RequestContext.getTimeZone()`方法获取用户的`TimeZone`。 Spring 的`ConversionService`注册的任何 Date/Time `Converter`和`Formatter`对象都会自动使用时区信息。

3.1.9.2 Header Resolver

此语言环境解析器检查 Client 端(例如，Web 浏览器)发送的请求中的`accept-language`Headers。通常，此头字段包含 Client 端 os 的语言环境。请注意，此解析器不支持时区信息。

3.1.9.3 Cookie Resolver

此语言环境解析器检查 Client 端上可能存在的`Cookie`，以查看是否指定了`Locale`或`TimeZone`。如果是这样，它将使用指定的详细信息。通过使用此语言环境解析器的属性，可以指定 Cookie 的名称以及最长期限。以下示例定义了`CookieLocaleResolver`：

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when
browser shuts down) -->
    <property name="cookieMaxAge" value="100000"/>

</bean>
```

下表描述了属性`CookieLocaleResolver`：

表 4. CookieLocaleResolver 属性

Property	Default	Description
`cookieName`	类别名称 LOCALE	Cookie 的名称
`cookieMaxAge`	Servlet 容器默认	Cookie 在 Client 端上保留的最长时间。如果指定了`-1`，则 cookie 将不会保留。它仅在 Client 端关闭浏览器之前可用。
`cookiePath`	/	将 Cookie 的可见性限制在您网站的特定部分。指定`cookiePath`时，该 cookie 仅对该路径及其下方的路径可见。

3.1.9.4 Session Resolver

`SessionLocaleResolver`可让您从可能与用户请求关联的会话中检索`Locale`和`TimeZone`。与`CookieLocaleResolver`相反，此策略将本地选择的语言环境设置存储在 Servlet 容器的`HttpSession`中。结果，这些设置对于每个会话都是临时的，因此在每个会话终止时会丢失。

请注意，与外部会话 Management 机制(例如 Spring Session 项目)没有直接关系。该`SessionLocaleResolver`对当前`HttpServletRequest`评估并修改了相应的`HttpSession`属性。

3.1.9.5 Locale Interceptor

您可以通过将 `LocaleChangeInterceptor` 添加到 `HandlerMapping` 定义之一来启用语言环境更改。它在请求中检测到一个参数，并相应地更改语言环境，从而在调度程序的应用程序上下文中在 `LocaleResolver` 上调用 `setLocale` 方法。下一个示例显示对包含参数 `siteLanguage` 的所有 `*.view` 资源的调用现在会更改语言环境。因此，例如，对 URL `http://www.sf.net/home.view?siteLanguage=nl` 的请求将站点语言更改为荷兰语。以下示例显示如何拦截语言环境：

```
<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
        class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

3.1.10 Themes

您可以应用 Spring Web MVC 框架主题来设置应用程序的整体外观，从而增强用户体验。主题是静态资源(通常是样式表和图像)的集合，这些资源会影响应用程序的视觉样式。

3.1.10.1 定义主题

要在 Web 应用程序中使用主题，必须设置 `org.springframework.ui.context.ThemeSource` 接口的实现。 `WebApplicationContext` 接口扩展了 `ThemeSource` ，但将其职责委托给专用的实现。默认情况下，委托是 `org.springframework.ui.context.support.ResourceBundleThemeSource` 实现，该实现从 Classpath 的根目录加载属性文件。要使用自定义 `ThemeSource` 实现或配置 `ResourceBundleThemeSource` 的基本名称前缀，可以在应用程序上下文中使用保留名称 `themeSource` 注册 Bean。 Web 应用程序上下文会自动检测到具有该名称的 bean 并使用它。

当您使用 `ResourceBundleThemeSource` 时，将在一个简单的属性文件中定义一个主题。属性文件列出了组成主题的资源，如以下示例所示：

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

属性的键是引用视图代码中主题元素的名称。对于 JSP，通常使用 `spring:theme` 自定义标签来完成此操作，该标签与 `spring:message` 标签非常相似。以下 JSP 片段使用上一示例中定义的主题来自定义外观：

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="<spring:theme code='styleSheet'/>"
type="text/css"/>
    </head>
    <body style="background=<spring:theme code='background'/>">
        ...
    </body>
</html>
```

默认情况下，`ResourceBundleThemeSource`使用空的基本名称前缀。结果，从 Classpath 的根加载属性文件。因此，您可以将`cool.properties`主题定义放在 Classpath 的根目录中(例如，在`/WEB-INF/classes`中)。 `ResourceBundleThemeSource`使用标准的 Java 资源束加载机制，允许主题的完全国际化。例如，我们可能有一个`/WEB-INF/classes/cool_nl.properties`，它引用了带有荷兰 Literals 的特殊背景图像。

3.1.10.2 Resolving Themes

定义主题后，如preceding section中所述，您可以决定使用哪个主题。 `DispatcherServlet`查找名为`themeResolver`的 bean，以找出要使用的`ThemeResolver`实现。主题解析器的工作方式与`LocaleResolver`大致相同。它可以检测用于特定请求的主题，还可以更改请求的主题。下表描述了 Spring 提供的主题解析器：

表 5. ThemeResolver 实现

Class	Description
`FixedThemeResolver`	选择通过使用`defaultThemeName`属性设置的固定主题。
`SessionThemeResolver`	主题在用户的 HTTP 会话中维护。每个会话只需设置一次，但在会话之间不会保留。
`CookieThemeResolver`	所选主题存储在 Client 端的 cookie 中。

Spring 还提供了一个`ThemeChangeInterceptor`，该主题可以使用简单的 request 参数在每个请求上更改主题。

3.1.11 Multipart 解析器

`org.springframework.web.multipart`包中的`MultipartResolver`是一种用于解析包括文件上传在内的 Multipart 请求的策略。有一种基于Commons FileUpload的实现，另一种基于 Servlet 3.0Multipart 请求解析。

要启用 Multipart 处理，您需要在`DispatcherServlet` Spring 配置中声明名称为`multipartResolver`的`MultipartResolver` bean。 `DispatcherServlet`检测到它并将其应用于传入的请求。收到 Content Type 为`multipart/form-data`的 POST 时，解析程序将解析内容并将当前`HttpServletRequest`包装为`MultipartHttpServletRequest`以提供对已解析部分的访问权限，除了将其公开为请求参数之外。

3.1.11.0.1 Apache Commons FileUpload

要使用 Apache Commons `FileUpload`，可以配置名称为`multipartResolver`的类型`CommonsMultipartResolver`的 bean。您还需要`commons-fileupload`作为对 Classpath 的依赖。

3.1.11.0.2 Servlet 3.0

需要通过 Servlet 容器配置启用 Servlet 3.0Multipart 解析。为此：

- 在 Java 中，在 Servlet 注册上设置`MultipartConfigElement`。
- 在`web.xml`中，将`"<multipart-config>"`部分添加到 Servlet 声明中。

以下示例显示了如何在 Servlet 注册上设置`MultipartConfigElement`：

```
public class AppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    // ...

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic
registration) {

        // Optionally also set maxFileSize, maxRequestSize,
fileSizeThreshold
        registration.setMultipartConfig(new
MultipartConfigElement("/tmp"));
    }

}
```

Servlet 3.0 配置到位后，您可以添加名称为`multipartResolver`的类型为`StandardServletMultipartResolver`的 bean。

3.1.12 Logging

springmvc中的调试级日志记录被设计成紧凑、最小化和人性化。它关注的是反复有用的高价值信息，而不是只有在调试特定问题时才有用的信息。

TRACE 级别的日志记录通常遵循与 DEBUG 相同的原则(例如，也不应是消防水带)，但可用于调试任何问题。此外，某些日志消息在 TRACE 和 DEBUG 上可能显示不同级别的详细信息。

3.1.12.1 Sensitive Data

调试和跟踪日志记录可能会记录敏感信息。这就是为什么默认情况下屏蔽请求参数和 Headers，并且必须通过`DispatcherServlet`上的`enableLoggingRequestDetails`属性显式启用它们的完整登录的原因。

以下示例显示了如何使用 Java 配置来执行此操作：

```
public class MyInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return ... ;
    }

    @Override
```

```
    protected Class<?>[] getServletConfigClasses() {
        return ... ;
    }

    @Override
    protected String[] getServletMappings() {
        return ... ;
    }

    @Override
    protected void customizeRegistration(Dynamic registration) {
        registration.setInitParameter("enableLoggingRequestDetails",
"true");
    }

}
```

3.2 Filters

`spring-web` 模块提供了一些有用的过滤器：

- Form Data
- Forwarded Headers
- Shallow ETag
- CORS

3.2.1 Form 数据

浏览器只能通过 HTTP GET 或 HTTP POST 提交表单数据，但非浏览器 Client 端也可以使用 HTTP PUT，PATCH 和 DELETE。 Servlet API 需要 `ServletRequest.getParameter*()` 个方法才能仅对 HTTP POST 支持表单字段访问。

`spring-web` 模块提供 `FormContentFilter` 来拦截 Content Type 为 `application/x-www-form-urlencoded` 的 HTTP PUT，PATCH 和 DELETE 请求，从请求的正文中读取表单数据，并包装 `ServletRequest` 以通过 `ServletRequest.getParameter*()` 系列方法使表单数据可用。

3.2.2 Forwarded Headers

当请求通过代理(例如负载平衡器)进行处理时，主机，端口和方案可能会更改，这使得从 Client 端角度创建指向正确的主机，端口和方案的链接带来了挑战。

RFC 7239定义 `Forwarded` HTTPHeaders，代理可用来提供有关原始请求的信息。还有其他非标准 Headers，包括 `X-Forwarded-Host`，`X-Forwarded-Port`，`X-Forwarded-Proto`，`X-Forwarded-Ssl` 和 `X-Forwarded-Prefix`。

`ForwardedHeaderFilter` 是一个 Servlet 过滤器，它根据 `Forwarded` Headers 修改请求的主机，端口和方案，然后删除这些 Headers。

对于转发的 Headers，存在安全方面的考虑，因为应用程序无法知道 Headers 是由代理添加的，还是由恶意 Client 端添加的。这就是为什么应配置信任边界处的代理以删除来自外部的不受信任的 `Forwarded` Headers 的原因。您还可以使用 `removeOnly=true` 配置 `ForwardedHeaderFilter`，在这种情况下，它将删除但不使用 Headers。

3.2.3 Shallow ETag

`ShallowEtagHeaderFilter`过滤器通过缓存写入响应的内容并从中计算 MD5 哈希值来创建“浅”ETag。Client 端下一次发送时，将执行相同的操作，但还会将计算值与`If-None-Match`请求 Headers 进行比较，如果两者相等，则返回 304(NOT_MODIFIED)。

此策略可节省网络带宽，但不能节省 CPU，因为必须为每个请求计算完整响应。如前所述，控制器级别的其他策略可以避免计算。参见HTTP Caching。

该过滤器具有一个`writeWeakETag`参数，该参数将过滤器配置为写入弱 ETag，类似于以下代码：`W/"02a2d595e6ed9a0b24f027f2b63b134d6"`(在RFC 7232 第 2.3 节中定义)。

3.2.4 CORS

Spring MVC 通过控制器上的注解 为 CORS 配置提供了细粒度的支持。但是，当与 Spring Security 一起使用时，我们建议您依赖内置的`CorsFilter`，该`CorsFilter`必须在 Spring Security 的过滤器链之前 Order。

3.3 Annotated Controllers 带注解 的控制器

Spring MVC 提供了基于注解 的编程模型，其中`@Controller`和`@RestController`组件使用注解 来表达请求映射，请求 Importing，异常处理等。带注解 的控制器具有灵活的方法签名，无需扩展 Base Class 或实现特定的接口。以下示例显示了由注解 定义的控制器：

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String handle(Model model) {
        model.addAttribute("message", "Hello World!");
        return "index";
    }
}
```

在前面的示例中，该方法接受`Model`并以`String`的形式返回视图名称，但是还存在许多其他选项，本章稍后将对其进行说明。

Tip：

spring.io上的指南和教程使用本节中介绍的基于注解 的编程模型。

3.3.1 Declaration

您可以使用 Servlet 的`WebApplicationContext`中的标准 Spring bean 定义来定义控制器 bean。`@Controller`原型允许自动检测，与 Spring 对在 Classpath 中检测`@Component`类并为其自动注册 Bean 定义的常规支持保持一致。它还充当带注解 类的构造型，表明其作为 Web 组件的作用。

要启用对此类`@Controller` bean 的自动检测，可以将组件扫描添加到 Java 配置中，如以下示例所示：

```java
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

下面的示例显示与前面的示例等效的 XML 配置：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example.web"/>

    <!-- ... -->

</beans>
```

`@RestController`是一个composed annotation，它本身会用`@Controller`和`@ResponseBody`进行元注解，以表示其每个方法都继承了类型级别`@ResponseBody`Comments 的控制器，因此直接将其写入响应主体(与视图分辨率和 HTML 模板渲染相比)。

AOP Proxies

在某些情况下，您可能需要在运行时用AOP代理来装饰控制器。一个例子是如果您选择在控制器上直接使用@Transactional注释。在这种情况下，特别是对于控制器，我们建议使用基于类的代理。这通常是控制器的默认选择。但是，如果控制器必须实现一个不是Spring上下文回调的接口（例如initializengbean、*Aware等），那么您可能需要显式地配置基于类的代理。例如，使用可以更改为，使用@EnableTransactionManagement可以更改为@EnableTransactionManagement（proxyTargetClass=true）。

3.3.2 Request Mapping 请求映射

您可以使用`@RequestMapping`注解将请求 Map 到控制器方法。它具有各种属性，可以通过 URL，HTTP 方法，请求参数，Headers 和媒体类型进行匹配。您可以在类级别使用它来表示共享的 Map，也可以在方法级别使用它来缩小到特定的端点 Map。

也有`@RequestMapping`的 HTTP 方法特定的快捷方式：

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

之所以提供快捷方式是Custom Annotations，是因为可以说，大多数控制器方法应该映射到特定的 HTTP 方法，而不是使用 `@RequestMapping`（默认情况下，该方法与所有 HTTP 方法匹配）。同时，在类级别仍需要 `@RequestMapping` 来表示共享映射。

以下示例具有类型和方法级别的 映射：

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

URI patterns

可以使用URL模式映射RequestMapping方法。有两种选择：

- PathPattern - 与URL路径匹配的预解析模式也预解析为PathContainer。此解决方案专为web使用而设计，它有效地处理编码和路径参数，并有效地进行匹配。
- AntPathMatcher - 根据字符串路径匹配字符串模式。这是Spring配置中使用的原始解决方案，用于选择类路径、文件系统和其他位置上的资源。它的效率较低，字符串路径输入对于有效地处理url的编码和其他问题是一个挑战。