网络上对这个问题的分析及解决不是很深入，大部分并不能解决问题，而且内容基本相同，拿来主义，把内容放在自己的博客上！

报错原因可能有两种情况：
1.请求头中没有设置Content-Type参数，或Content-Type参数值不是application/json；
2.请求头中正确设置了Content-Type参数及参数值，但是在项目jar依赖中（pom.xml或build.gradle）没有添加处理json字符串的处理类，如果Spring MVC框架在启动的时候，检查com.fasterxml.jackson.databind.ObjectMapper和com.fasterxml.jackson.core.JsonGenerator有一个不存在或不能加载，则不会注册MappingJackson2HttpMessageConverter，这个类使用Jackson将json请求参数转成相应的方法参数；同样检查com.google.gson.Gson，如果不存在或不能加载，则不会注册GsonHttpMessageConverter，这个类使用Gson将json请求参数转成相应的方法参数；如果依赖的Jackson和Gson都没有被添加或不能加载，则SpringMVC将找不到对应的参数处理类。

源码分析
在使用SpringMVC的时候，都会添加<mvc:annotation-driven />注解，这个注解下有很多可以配置的扩展参数，有兴趣的可以研究一下。有这个注解，就必定有对应的注解解析，查看NamespaceHandler接口的实现类，发现有一个MvcNamespaceHandler。

```
public class MvcNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
        registerBeanDefinitionParser("default-servlet-handler", new DefaultServletBeanDefinitionParser());
        registerBeanDefinitionParser("interceptors", new InterceptorsBeanDefinitionParser());
        registerBeanDefinitionParser("resources", new ResourcesBeanDefinitionParser());
        registerBeanDefinitionParser("view-controller", new ViewControllerBeanDefinitionParser());
        registerBeanDefinitionParser("redirect-view-controller", new ViewControllerBeanDefinitionParser());
        registerBeanDefinitionParser("status-controller", new ViewControllerBeanDefinitionParser());
        registerBeanDefinitionParser("view-resolvers", new ViewResolversBeanDefinitionParser());
        registerBeanDefinitionParser("tiles-configurer", new TilesConfigurerBeanDefinitionParser());
        registerBeanDefinitionParser("freemarker-configurer", new FreeMarkerConfigurerBeanDefinitionParser());
        registerBeanDefinitionParser("velocity-configurer", new VelocityConfigurerBeanDefinitionParser());
        registerBeanDefinitionParser("groovy-configurer", new GroovyMarkupConfigurerBeanDefinitionParser());
    }
}
```

annotation-driven注解做了什么，直接看AnnotationDrivenBeanDefinitionParser类。这个类中主要的就是parse方法，这个方法中做了很多重要的事，如对一些可扩展的参数进行了解析注册，这些不是本篇的重点，有兴趣的可以研究一下，关注重点代码。

```
ManagedList<?> messageConverters = getMessageConverters(element, source, parserContext);
ManagedList<?> argumentResolvers = getArgumentResolvers(element, parserContext);
ManagedList<?> returnValueHandlers = getReturnValueHandlers(element, parserContext);
String asyncTimeout = getAsyncTimeout(element);
RuntimeBeanReference asyncExecutor = getAsyncExecutor(element);
ManagedList<?> callableInterceptors = getCallableInterceptors(element, source, parserContext);
ManagedList<?> deferredResultInterceptors = getDeferredResultInterceptors(element, source, parserContext);

RootBeanDefinition handlerAdapterDef = new RootBeanDefinition(RequestMappingHandlerAdapter.class);
handlerAdapterDef.setSource(source);
handlerAdapterDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
handlerAdapterDef.getPropertyValues().add("contentNegotiationManager", contentNegotiationManager);
handlerAdapterDef.getPropertyValues().add("webBindingInitializer", bindingDef);
handlerAdapterDef.getPropertyValues().add("messageConverters", messageConverters);
addRequestBodyAdviceBeans(handlerAdapterDef);
```

代码中的messageConverters是消息转换器集合，里面包含了对json、xml、atom、rss格式报文的转换。接着，把messageConverters添加到RequestMappingHandlerAdapter中，RequestMappingHandlerAdapter是处理@RequestMapping注解的HandlerAdapter，简单说就是标注了@RequestMapping注解的Controller，是经过RequestMappingHandlerAdapter进行调用的，messageConverters是它的一个属性，代码如下。

```
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    private List<HandlerMethodArgumentResolver> customArgumentResolvers;

    private HandlerMethodArgumentResolverComposite argumentResolvers;

    private HandlerMethodArgumentResolverComposite initBinderArgumentResolvers;

    private List<HandlerMethodReturnValueHandler> customReturnValueHandlers;

    private HandlerMethodReturnValueHandlerComposite returnValueHandlers;

    private List<ModelAndViewResolver> modelAndViewResolvers;

    private ContentNegotiationManager contentNegotiationManager = new ContentNegotiationManager();

    private List<HttpMessageConverter<?>> messageConverters;

    private List<Object> requestResponseBodyAdvice = new ArrayList<Object>();
```

继续看AnnotationDrivenBeanDefinitionParser类，分析上图红框中的ManagedList<?> messageConverters = getMessageConverters(element, source, parserContext)，深入这个getMessageConverters方法。

```
protected ManagedList<?> getMessageConverters(Element element, Object source, ParserContext parserContext) {
    Element converterElement = DomUtils.getChildElementByTagName(element, "message-converters");
    ManagedList<? super Object> messageConverters = new ManagedList<Object>();
    if (converterElement != null) {
        messageConverters.setSource(source);
        for (Element beanElement : DomUtils.getChildElementsByTagName(converterElement, "bean", "ref")) {
            Object beanDef = parserContext.getDelegate().parsePropertySubElement(beanElement, null);
            messageConverters.add(beanDef);
        }
    }

    if (converterElement == null || Boolean.valueOf(converterElement.getAttribute("register-defaults"))) {
        messageConverters.setSource(source);
        messageConverters.add(createConverterDefinition(ByteArrayHttpMessageConverter.class, source));

        RootBeanDefinition stringConverterDef = createConverterDefinition(StringHttpMessageConverter.class, source);
        stringConverterDef.getPropertyValues().add("writeAcceptCharset", false);
        messageConverters.add(stringConverterDef);

        messageConverters.add(createConverterDefinition(ResourceHttpMessageConverter.class, source));
        messageConverters.add(createConverterDefinition(SourceHttpMessageConverter.class, source));
        messageConverters.add(createConverterDefinition(AllEncompassingFormHttpMessageConverter.class, source));

        if (romePresent) {
            messageConverters.add(createConverterDefinition(AtomFeedHttpMessageConverter.class, source));
            messageConverters.add(createConverterDefinition(RssChannelHttpMessageConverter.class, source));
        }

        if (jaxb2Present || jackson2Present) {
            messageConverters.add(createConverterDefinition(MappingJackson2HttpMessageConverter.class, source));
        }
        else if (jackson2Present) {
            messageConverters.add(createConverterDefinition(MappingJackson2XmlHttpMessageConverter.class, source));
        }

        if (jaxb2Present) {
            messageConverters.add(createConverterDefinition(Jaxb2RootElementHttpMessageConverter.class, source));
        }

        if (jackson2Present) {
            messageConverters.add(createConverterDefinition(MappingJackson2HttpMessageConverter.class, source));
        }
        else if (gsonPresent) {
            messageConverters.add(createConverterDefinition(GsonHttpMessageConverter.class, source));
        }
    }
    return messageConverters;
}
```

romePresent、jaxb2Present、jackson2Present、jackson2XmlPresent、gsonPresent为true则将对应的转换器包装成BeanDefinition，然后将其添加到messageConverters集合中。这几个布尔变量的值在AnnotationDrivenBeanDefinitionParser类的开头处就赋值了。

```
class AnnotationDrivenBeanDefinitionParser implements BeanDefinitionParser {

    public static final String CONVERSION_SERVICE_MANAGER_BEAN_NAME = "mvcConversionServiceManager";

    private static final boolean javaxValidationPresent =
            ClassUtils.isPresent("javax.validation.Validator", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

    private static boolean romePresent =
            ClassUtils.isPresent("com.rometools.rome.feed.WireFeed", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

    private static final boolean jaxb2Present =
            ClassUtils.isPresent("javax.xml.bind.Binder", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

    private static final boolean jackson2Present =
            ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper", AnnotationDrivenBeanDefinitionParser.class.getClassLoader()) &&
            ClassUtils.isPresent("com.fasterxml.jackson.core.JsonGenerator", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

    private static final boolean jackson2XmlPresent =
            ClassUtils.isPresent("com.fasterxml.jackson.dataformat.xml.XmlMapper", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

    private static final boolean gsonPresent =
            ClassUtils.isPresent("com.google.gson.Gson", AnnotationDrivenBeanDefinitionParser.class.getClassLoader());
```

如果相应的实现类存在并且可以被加载，则对应的布尔变量值为true，否则为false。也就是说，如果SpringMVC框架在启动的时候，检查com.fasterxml.jackson.databind.ObjectMapper和com.fasterxml.jackson.core.JsonGenerator有一个不存在或不能加载，则不会注册MappingJackson2HttpMessageConverter，这个类使用Jackson将json请求参数转成相应的方法参数；同样检查com.google.gson.Gson，如果不存在或不能加载，则不会注册GsonHttpMessageConverter，这个类使用Gson将json请求参数转成相应的方法参数；如果依赖的Jackson和Gson都没有被添加或不能加载，则SpringMVC将找不到json参数转换类，也就没办法处理。

如果配置了json参数转换处理类，SpringMVC框架将根据请求头中的Content-Type参数遍历messageConverters，选择匹配的转换器类，进行参数转换。如果Content-Type参数值类型是messageConverters中不支持的，那么就没办法做转换。

总结：
首先，SpringMVC框架在启动的时候会遍历Spring容器中的所有bean，对标注了@Controller或@RequestMapping注解的类中方法进行遍历，将类和方法上的@RequestMapping注解值进行合并，使用@RequestMapping注解的相关参数值(如value、method等)封装一个RequestMappingInfo，将这个Controller实例、方法及方法参数信息(类型、注解等)封装到HandlerMethod中，然后以RequestMappingInfo为key，HandlerMethod为value存到一个以Map为结构的handlerMethods中。

接着，将@RequestMapping注解中的value(即请求路径)值取出，即url，然后以url为key，以RequestMappingInfo为value，存到一个以Map为结构的urlMap属性中。

客户端发起请求的时候，根据请求的URL到urlMap中查找，找到RequestMappingInfo，然后根据RequestMappingInfo到handlerMethods中查找，找到对应的HandlerMethod，接着将HandlerMethod封装到HandlerExecutionChain；接着遍历容器中所有HandlerAdapter实现类，找到支持这次请求的HandlerAdapter，如RequestMappingHandlerAdapter，然后执行SpringMVC拦截器的前置方法(preHandle方法)，然后对请求参数解析及转换，这里主要根据HandlerMethod中封装的参数信息(方法参数上的注解)来遍历argumentResolvers(List结构，存储了HandlerMethodArgumentResolver接口实现类，不同实现类，实现对不同注解参数的解析，如RequestResponseBodyMethodProcessor可以实现对@RequestBody和@ResponseBody参数的解析)，找到支持这个注解的HandlerMethodArgumentResolver实现类，然后解析请求参数。

插播一下请求参数的解析及转换，下图是HandlerMethodArgumentResolver接口的实现类。

从上图中可以看到很多常见注解参数的解析类，这里分析RequestResponseBodyMethodProcessor，其它处理类感兴趣的可以自己研究一下。RequestResponseBodyMethodProcessor会从请求头中获取Content-Type参数值，例如application/json，然后遍历messageConverters，查找能够处理这种Content-Type的转换器类，如果messageConverters中有可以处理application/json请求的处理类，如Jackson或Gson，则使用Jackson或Gson对请求体中的参数进行读取转换，转换成具体方法参数类型，下面是Jackson具体的处理代码。

```
private Object readJavaType(JavaType javaType, HttpInputMessage inputMessage) {
    try {
        return this.objectMapper.readValue(inputMessage.getBody(), javaType);
    }
    catch (IOException ex) {
        throw new HttpMessageNotReadableException("Could not read JSON: " + ex.getMessage(), ex);
    }
}
```

如果messageConverters没有匹配的处理类，那就会报415。

最后，(使用反射)调用具体Controller的对应方法返回一个ModelAndView对象，执行拦截器的后置方法(postHandle方法)，然后对返回的结果进行处理，最后执行afterCompletion方法。