



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

## 【死磕 Spring】—— IOC 之深入分析 Bean 的类型转换体系 (<http://cmsblogs.com/?p=3983>)

2019-01-20 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(5920) 评论(0)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

我们知道不管 bean 对象里面的属性是什么类型，他们都是通过 XML、Properties 或者其他方式来配置这些属性对象类型的。在 Spring 容器加载过程中，这些属性都是以 String 类型加载进容器的，但是最终都需要将这些 String 类型的属性转换 Bean 对象属性所对应真正的类型，要想完成这种由字符串到具体对象的转换，就需要这种转换规则相关的信息，而这些信息以及转换过程由 Spring 类型转换体系来完成。

我们依然以 xml 为例，在 Spring 容器加载阶段，容器将 xml 文件中定义的 <bean> 解析为 BeanDefinition，BeanDefinition 中存储着我们定义一个 bean 需要的所有信息，包括属性，这些属性是以 String 类型的存储的。当用户触发 Bean 实例化阶段时，Spring 容器会将这些属性转换为这些属性真正对应的类型。我们知道在 bean 实例化阶段，属性的注入是在实例化 bean 阶段的属性注入阶段，即 populateBean() 方法。在 populateBean() 中会将 BeanDefinition 中定义的属性值翻译为 PropertyValue 然后调用 applyPropertyValues() 进行属性应用。其中 PropertyValue 用于保存单个 bean 属性的信息和值的对象。在 applyPropertyValues() 中会调用 convertForProperty() 进行属性转换，如下：

```
private Object convertForProperty(
    @Nullable Object value, String propertyName, BeanWrapper bw, TypeConverter converter) {

    if (converter instanceof BeanWrapperImpl) {
        return ((BeanWrapperImpl) converter).convertForProperty(value, propertyName);
    }
    else {
        PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
        MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);
        return converter.convertIfNecessary(value, pd.getPropertyType(), methodParam);
    }
}
```

若 TypeConverter 为 BeanWrapperImpl 类型，则使用 BeanWrapperImpl 来进行类型转换，这里主要是因为 BeanWrapperImpl 实现了 PropertyEditorRegistry 接口。否则则调用 TypeConverter 的 convertIfNecessary() 进行类型转换。TypeConverter 是定义类型转换方法的接口，通常情况下与 PropertyEditorRegistry 配合使用实现类型转换。关于 BeanWrapperImpl 小编后续专门出文分析它。

convertIfNecessary() 的实现者有两个：DataBinder 和 TypeConverterSupport，其中 DataBinder 主要用于参数绑定（熟悉 Spring MVC 的都应该知道这个类），TypeConverterSupport 则是 TypeConverter 的基本实现，使用的是 package-private 策略。所以这里我们只需要关注 TypeConverterSupport 的 convertIfNecessary()，如下：

```
public <T> T convertIfNecessary(@Nullable Object value, @Nullable Class<T> requiredType, @Nullable MethodParameter methodParam)
    throws TypeMismatchException {

    return doConvert(value, requiredType, methodParam, null);
}

private <T> T doConvert(@Nullable Object value, @Nullable Class<T> requiredType,
    @Nullable MethodParameter methodParam, @Nullable Field field) throws TypeMismatchException {

    Assert.state(this.typeConverterDelegate != null, "No TypeConverterDelegate");
    try {
        if (field != null) {
            return this.typeConverterDelegate.convertIfNecessary(value, requiredType, field);
        }
        else {
            return this.typeConverterDelegate.convertIfNecessary(value, requiredType, methodParam);
        }
    }
    catch (ConverterNotFoundException | IllegalStateException ex) {
        throw new ConversionNotSupportedException(value, requiredType, ex);
    }
    catch (ConversionException | IllegalArgumentException ex) {
        throw new TypeMismatchException(value, requiredType, ex);
    }
}
```

我们一直往下跟会跟踪到 TypeConverterDelegate 的 convertIfNecessary()，会发现如下代码段：

```
// No custom editor but custom ConversionService specified?
ConversionService conversionService = this.propertyEditorRegistry.getConversionService();
if (editor == null && conversionService != null && newValue != null && typeDescriptor != null) {
    TypeDescriptor sourceTypeDesc = TypeDescriptor.forObject(newValue);
    if (conversionService.canConvert(sourceTypeDesc, typeDescriptor)) {
        try {
            return (T) conversionService.convert(newValue, sourceTypeDesc, typeDescriptor);
        }
        catch (ConversionFailedException ex) {
            // fallback to default conversion logic below
            conversionAttemptEx = ex;
        }
    }
}
```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15388823317115.jpg>)

如果没有自定义的编辑器则使用 ConversionService。ConversionService 是 Spring 自 3 后推出来用来替代 PropertyEditor 转换模式的转换体系，接口定义如下：

```
public interface ConversionService {
```



```
    boolean canConvert(@Nullable Class<?> sourceType, Class<?> targetType);
```

```
    boolean canConvert(@Nullable TypeDescriptor sourceType, TypeDescriptor targetType);
```

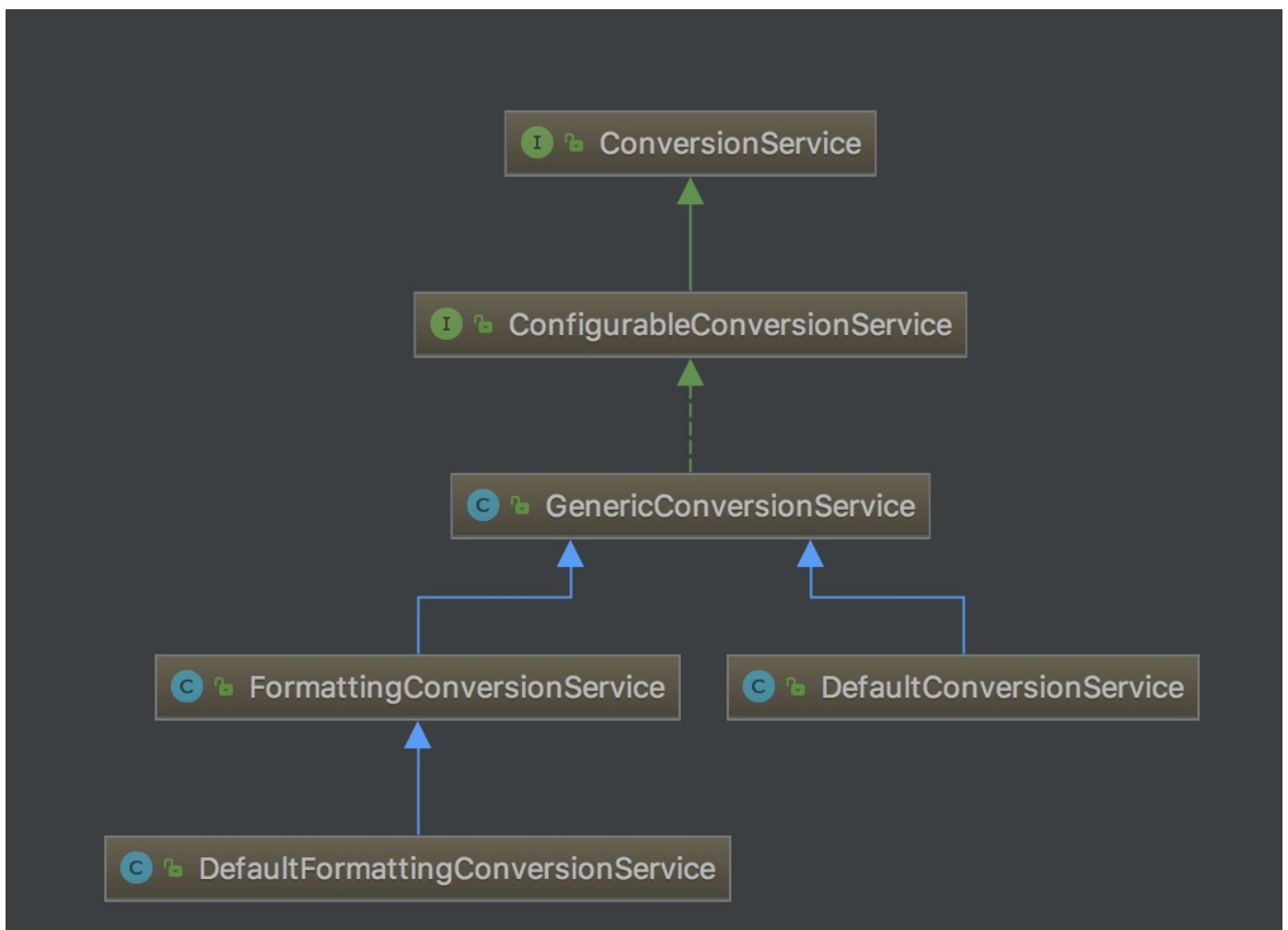
```
    @Nullable
```

```
    <T> T convert(@Nullable Object source, Class<T> targetType);
```

```
    @Nullable
```

```
    Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

其 UML 类图如下：



(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15388832533588.jpg>)

- **ConfigurableConversionService**：ConversionService 的配置接口，继承 ConversionService 和 ConverterRegistry 两个接口，用于合并他们两者的操作，以便于通过 add 和 remove 的方式添加和删除转换器。
- **GenericConversionService**：ConversionService 接口的基础实现，适用于大部分条件下的转换工作，通过 ConfigurableConversionService 接口间接地将 ConverterRegistry 实现为注册 API。
- **DefaultConversionService**：ConversionService 接口的默认实现，适用于大部分条件下的转换工作。

回归到 `convertIfNecessary()`，在该方法中如果没有自定义的属性编辑器则调用 `ConversionService` 接口的 `convert()`，方法定义如下：

```
Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType, TypeDescriptor targetType);
```

- `source`：要转换的源对象，可以为 `null`
- `sourceType`：`source` 的类型的上下文，如果 `source` 为 `null`，则可以为 `null`
- `targetType`：`source` 要转换的类型的上下文。

`convert()` 将给定的源对象 `source` 转换为指定的 `targetType`。`TypeDescriptors` 提供有关发生转换的源位置和目标位置的附加上下文，通常是对象字段或属性位置。方法由子类 `GenericConversionService` 实现：

```
public Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType, TypeDescriptor targetType) {
    // 删掉 if，其实就是上面的 null 判断
    GenericConverter converter = getConverter(sourceType, targetType);
    if (converter != null) {
        Object result = ConversionUtils.invokeConverter(converter, source, sourceType, targetType);
        return handleResult(sourceType, targetType, result);
    }
    return handleConverterNotFound(source, sourceType, targetType);
}
```

首先根据 `sourceType` 和 `targetType` 调用 `getConverter()` 获取 `GenericConverter` 对象 `converter`，如果 `converter` 为 `null`，则调用 `handleConverterNotFound()`，否则调用 `handleResult()` 方法。`getConverter()` 如下：

```
protected GenericConverter getConverter(TypeDescriptor sourceType, TypeDescriptor targetType) {
    ConverterCacheKey key = new ConverterCacheKey(sourceType, targetType);
    GenericConverter converter = this.converterCache.get(key);
    if (converter != null) {
        return (converter != NO_MATCH ? converter : null);
    }

    converter = this.converters.find(sourceType, targetType);
    if (converter == null) {
        converter = getDefaultConverter(sourceType, targetType);
    }

    if (converter != null) {
        this.converterCache.put(key, converter);
        return converter;
    }

    this.converterCache.put(key, NO_MATCH);
    return null;
}
```

这段代码意图非常明确，从 converterCache 缓存中获取，如果存在返回，否则从 converters 中获取，然后加入到 converterCache 缓存中。converterCache 和 converters 是 GenericConversionService 维护的两个很重要的对象，其中 converterCache 用于存储 GenericConverter，converters 对象为 GenericConversionService 的内部类。

```
private final Converters converters = new Converters();  
private final Map<ConverterCacheKey, GenericConverter> converterCache = new ConcurrentReferenceHashMap<>(64);
```

Converters 用于管理所有注册的转换器，其内部维护一个 Set 和 Map 的数据结构用于管理转换器，如下：

```
private final Set<GenericConverter> globalConverters = new LinkedHashSet<>();  
  
private final Map<ConvertiblePair, ConvertersForPair> converters = new LinkedHashMap<>(36);
```

同时提供了相应的方法（如 add、remove）操作这两个集合。在 getConverter() 中如果缓存 converterCache 中不存在，则调用 Converters 对象的 find() 方法获取相应的 GenericConverter，如下：

```

public GenericConverter find(TypeDescriptor sourceType, TypeDescriptor targetType) {
    // Search the full type hierarchy
    List<Class<?>> sourceCandidates = getClassHierarchy(sourceType.getType());
    List<Class<?>> targetCandidates = getClassHierarchy(targetType.getType());
    for (Class<?> sourceCandidate : sourceCandidates) {
        for (Class<?> targetCandidate : targetCandidates) {
            ConvertiblePair convertiblePair = new ConvertiblePair(sourceCandidate, targetCandidate);
            GenericConverter converter = getRegisteredConverter(sourceType, targetType, convertiblePair);
            if (converter != null) {
                return converter;
            }
        }
    }
    return null;
}

private GenericConverter getRegisteredConverter(TypeDescriptor sourceType,
        TypeDescriptor targetType, ConvertiblePair convertiblePair) {

    // Check specifically registered converters
    ConvertersForPair convertersForPair = this.converters.get(convertiblePair);
    if (convertersForPair != null) {
        GenericConverter converter = convertersForPair.getConverter(sourceType, targetType);
        if (converter != null) {
            return converter;
        }
    }
    // Check ConditionalConverters for a dynamic match
    for (GenericConverter globalConverter : this.globalConverters) {
        if (((ConditionalConverter) globalConverter).matches(sourceType, targetType)) {
            return globalConverter;
        }
    }
    return null;
}

```

在 `find()` 中会根据 `sourceType` 和 `targetType` 去查询 `Converters` 中维护的 `Map` 中是否包括支持的注册类型，如果存在返回 `GenericConverter`，如果没有存在返回 `null`。

当得到 `GenericConverter` 后，则调用其 `convert()` 进行类型转换。

```
Object convert(@Nullable Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
```

到这里我们就可以得到 `bean` 属性定义的真正类型了。

## GenericConverter 接口

`GenericConverter` 是一个转换接口，一个用于在两种或更多种类型之间转换的通用型转换器接口。它是 `Converter SPI` 体系中最灵活的，也是最复杂的接口，灵活性在于 `GenericConverter` 可以支持在多个源/目标类型对之间进行转换，同时也可以类型转换过程中访问源/目标字段上下文。由于该接口足够复杂，所有当



更简单的 Converter 或 ConverterFactory 接口足够使用时，通常不应使用此接口。其定义如下：



```
public interface GenericConverter {

    @Nullable
    Set<ConvertiblePair> getConvertibleTypes();

    @Nullable
    Object convert(@Nullable Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

GenericConverter 的子类有这么多（看类名就知道是干嘛的了）：




A screenshot of a list of subclasses of GenericConverter in the Spring Framework. The list is displayed in a dark-themed IDE. Each item is preceded by a blue circular icon containing a white 'C'. The items are as follows:

- AnnotationParserConverter in FormattingConversionService (org.springframework.format.support)
- AnnotationPrinterConverter in FormattingConversionService (org.springframework.format.support)
- ArrayToArrayConverter (org.springframework.core.convert.support)
- ArrayToCollectionConverter (org.springframework.core.convert.support)
- ArrayToObjectConverter (org.springframework.core.convert.support)
- ArrayToStringConverter (org.springframework.core.convert.support)
- ByteBufferConverter (org.springframework.core.convert.support)
- CollectionToArrayConverter (org.springframework.core.convert.support)
- CollectionToCollectionConverter (org.springframework.core.convert.support)
- CollectionToObjectConverter (org.springframework.core.convert.support)
- CollectionToStringConverter (org.springframework.core.convert.support)
- ConverterAdapter in GenericConversionService (org.springframework.core.convert.support)
- ConverterFactoryAdapter in GenericConversionService (org.springframework.core.convert.support)
- FallbackObjectToStringConverter (org.springframework.core.convert.support)
- IdToEntityConverter (org.springframework.core.convert.support)
- MapToMapConverter (org.springframework.core.convert.support)
- NoOpConverter in GenericConversionService (org.springframework.core.convert.support)
- ObjectToArrayConverter (org.springframework.core.convert.support)
- ObjectToCollectionConverter (org.springframework.core.convert.support)
- ObjectToObjectConverter (org.springframework.core.convert.support)
- ObjectToOptionalConverter (org.springframework.core.convert.support)
- ParserConverter in FormattingConversionService (org.springframework.format.support)
- PrinterConverter in FormattingConversionService (org.springframework.format.support)
- StreamConverter (org.springframework.core.convert.support)
- StringToArrayConverter (org.springframework.core.convert.support)
- StringToCollectionConverter (org.springframework.core.convert.support)

The last item, StringToCollectionConverter, is highlighted with a blue background.

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15388984648307.jpg>)

我们看一个子类的实现 StringToArrayConverter，该子类将逗号分隔的 String 转换为 Array。如下：

```

public Object convert(@Nullable Object source, TypeDescriptor sourceType, TypeDescriptor targetType)
{
    if (source == null) {
        return null;
    }
    String string = (String) source;
    String[] fields = StringUtils.commaDelimitedListToStringArray(string);
    TypeDescriptor targetElementType = targetType.getElementTypeDescriptor();
    Assert.state(targetElementType != null, "No target element type");
    Object target = Array.newInstance(targetElementType.getType(), fields.length);
    for (int i = 0; i < fields.length; i++) {
        String sourceElement = fields[i];
        Object targetElement = this.conversionService.convert(sourceElement.trim(), sourceType, targetElementType);
        Array.set(target, i, targetElement);
    }
    return target;
}

```

在类型转换体系中，Spring 提供了非常多的类型转换器，除了上面的 `GenericConverter`，还有 `Converter`、`ConditionalConverter`、`ConverterFactory`。

## Converter

`Converter` 是一个将 S 类型的源对象转换为 T 类型的目标对象的转换器。该接口是线程安全的，所以可以共享。

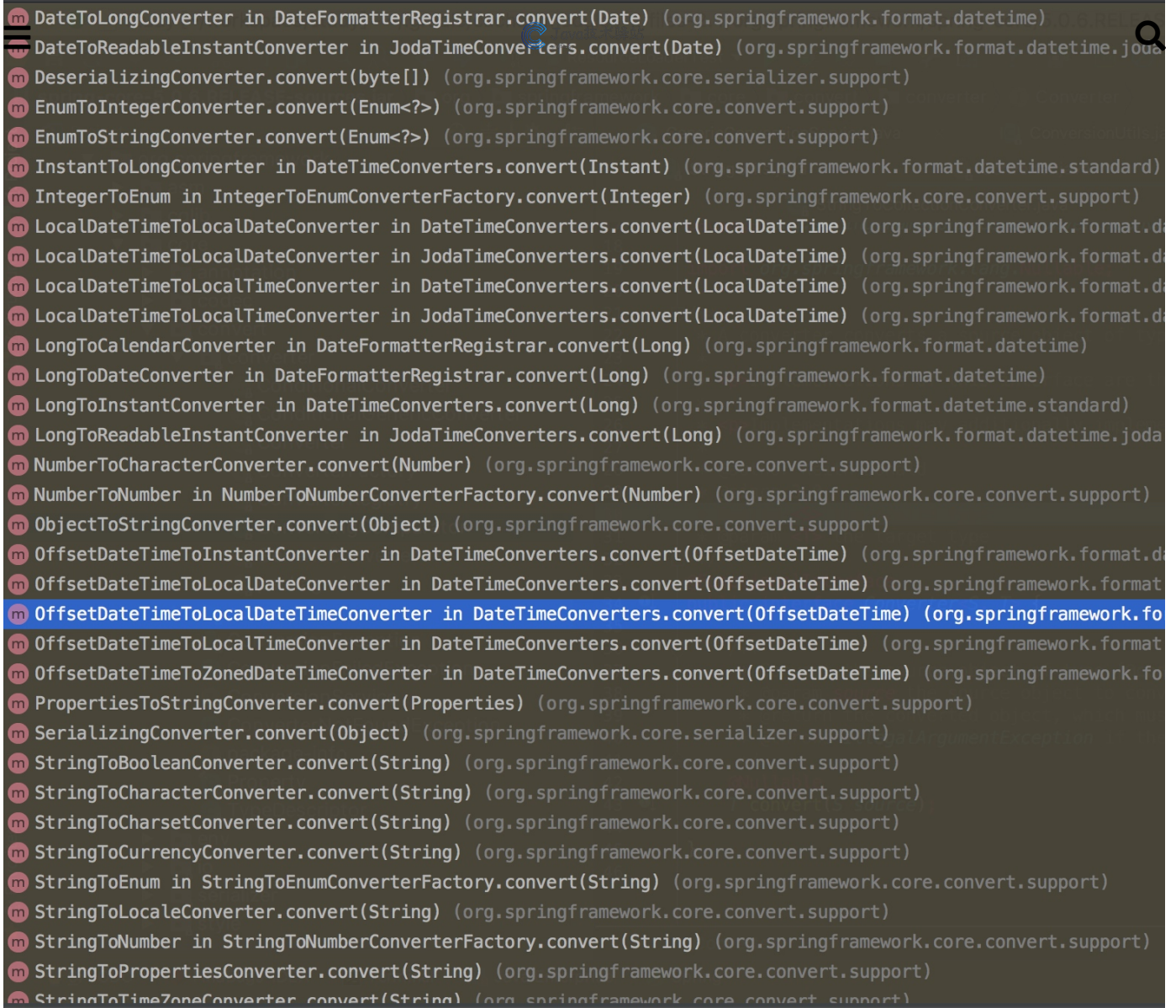
```

public interface Converter<S, T> {
    @Nullable
    T convert(S source);
}

```

子类如下：





```

DateToLongConverter in DateFormatterRegistrar.convert(Date) (org.springframework.format.datetime)
DateToReadableInstantConverter in JodaTimeConverters.convert(Date) (org.springframework.format.datetime.joda)
DeserializingConverter.convert(byte[]) (org.springframework.core.serializer.support)
EnumToIntegerConverter.convert(Enum<?>) (org.springframework.core.convert.support)
EnumToStringConverter.convert(Enum<?>) (org.springframework.core.convert.support)
InstantToLongConverter in DateTimeConverters.convert(Instant) (org.springframework.format.datetime.standard)
IntegerToEnum in IntegerToEnumConverterFactory.convert(Integer) (org.springframework.core.convert.support)
LocalDateTimeToLocalDateConverter in DateTimeConverters.convert(LocalDateTime) (org.springframework.format.datetime)
LocalDateTimeToLocalDateConverter in JodaTimeConverters.convert(LocalDateTime) (org.springframework.format.datetime)
LocalDateTimeToLocalTimeConverter in DateTimeConverters.convert(LocalDateTime) (org.springframework.format.datetime)
LocalDateTimeToLocalTimeConverter in JodaTimeConverters.convert(LocalDateTime) (org.springframework.format.datetime)
LongToCalendarConverter in DateFormatterRegistrar.convert(Long) (org.springframework.format.datetime)
LongToDateConverter in DateFormatterRegistrar.convert(Long) (org.springframework.format.datetime)
LongToInstantConverter in DateTimeConverters.convert(Long) (org.springframework.format.datetime.standard)
LongToReadableInstantConverter in JodaTimeConverters.convert(Long) (org.springframework.format.datetime.joda)
NumberToCharacterConverter.convert(Number) (org.springframework.core.convert.support)
NumberToNumber in NumberToNumberConverterFactory.convert(Number) (org.springframework.core.convert.support)
ObjectToStringConverter.convert(Object) (org.springframework.core.convert.support)
OffsetDateTimeToInstantConverter in DateTimeConverters.convert(OffsetDateTime) (org.springframework.format.datetime)
OffsetDateTimeToLocalDateConverter in DateTimeConverters.convert(OffsetDateTime) (org.springframework.format.datetime)
OffsetDateTimeToLocalDateTimeConverter in DateTimeConverters.convert(OffsetDateTime) (org.springframework.format.datetime)
OffsetDateTimeToLocalTimeConverter in DateTimeConverters.convert(OffsetDateTime) (org.springframework.format.datetime)
OffsetDateTimeToZonedDateTimeConverter in DateTimeConverters.convert(OffsetDateTime) (org.springframework.format.datetime)
PropertiesToStringConverter.convert(Properties) (org.springframework.core.convert.support)
SerializingConverter.convert(Object) (org.springframework.core.serializer.support)
StringToBooleanConverter.convert(String) (org.springframework.core.convert.support)
StringToCharacterConverter.convert(String) (org.springframework.core.convert.support)
StringToCharsetConverter.convert(String) (org.springframework.core.convert.support)
StringToCurrencyConverter.convert(String) (org.springframework.core.convert.support)
StringToEnum in StringToEnumConverterFactory.convert(String) (org.springframework.core.convert.support)
StringToLocaleConverter.convert(String) (org.springframework.core.convert.support)
StringToNumber in StringToNumberConverterFactory.convert(String) (org.springframework.core.convert.support)
StringToPropertiesConverter.convert(String) (org.springframework.core.convert.support)
StringToTimeZoneConverter.convert(String) (org.springframework.core.convert.support)

```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15389004851676.jpg>)

## ConditionalConverter

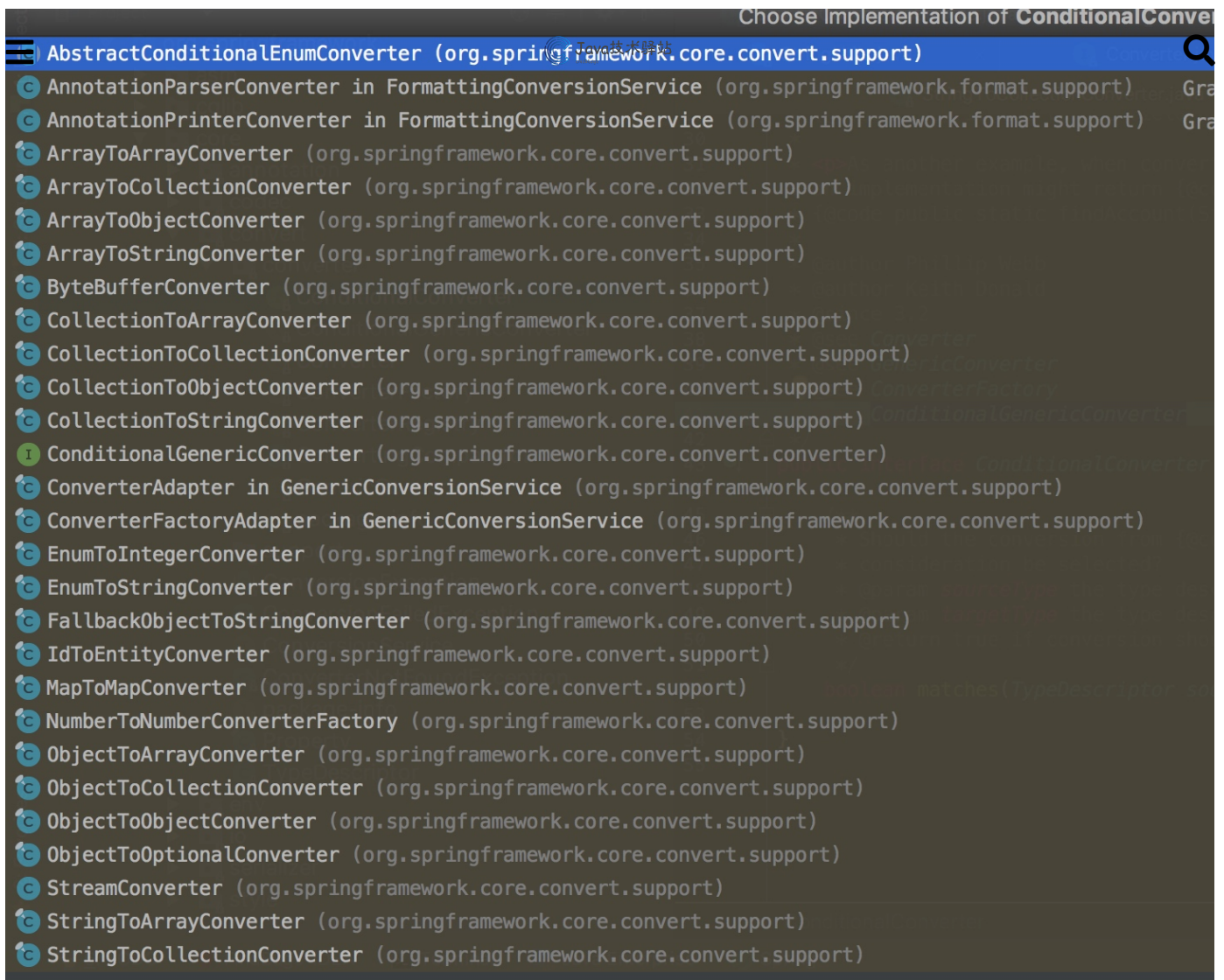
ConditionalConverter 接口用于表示有条件的类型转换，通过转入的sourceType 与 targetType 判断转换能否匹配，只有可匹配的转换才会调用convert 方法进行转换，如下：

```

public interface ConditionalConverter {
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}

```

ConditionalConverter 的子类如下：



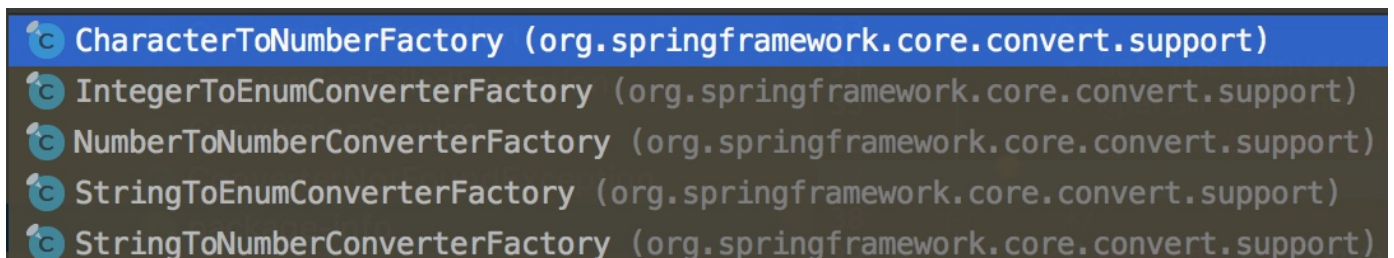
(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15389064279138.jpg>)

## ConverterFactory

一个用于“远程”转换的转换工厂，可以将对象从 S 转换为 R 的子类型。

```
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

子类如下：



(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15389071818355.jpg>)

四种不同的转换器承载着不同的转换过程：

- Converter：用于 1:1 的 source -> target 类型转换



- ConverterFactory: 用于 1:N 的 source -> target 类型转换
- GenericConverter 用于 N:N 的 source -> target 类型转换
- ConditionalConverter: 有条件的 source -> target 类型转换

## GenericConversionService

转换器介绍完了，我们再次回归到 ConversionService 接口中去，该接口定义了两类方法 canConvert() 和 convert()，其中 canConvert() 用于判 sourceType 能否转成 targetType，而 convert() 用于将 source 转成转入的 TargetType 类型实例。这两类方法都是在 GenericConversionService 中实现。类 GenericConversionService 实现 ConfigurableConversionService 接口，而 ConfigurableConversionService 接口继承 ConversionService 和 ConverterRegistry。ConverterRegistry 提供了类型转换器的管理功能，他提供了四个 add 和一个 remove 方法，支持注册/删除相应的类型转换器。GenericConversionService 作为一个基础实现类，它即支持了不同类型之间的转换，也对各类型转换器进行管理，主要是通过一个 Map 类型的 converterCache 和一个内部类 Converters。在上面已经分析了 GenericConversionService 执行类型转换的过程 cover()，下面我们就一个 addConverter() 来看看它是如何完成转换器的注入工作的。

```
public void addConverter(Converter<?, ?> converter) {
    ResolvableType[] typeInfo = getRequiredTypeInfo(converter.getClass(), Converter.class);
    if (typeInfo == null && converter instanceof DecoratingProxy) {
        typeInfo = getRequiredTypeInfo(((DecoratingProxy) converter).getDecoratedClass(), Converter.class);
    }
    if (typeInfo == null) {
        throw new IllegalArgumentException("Unable to determine source type <S> and target type <T> for your " +
            "Converter [" + converter.getClass().getName() + "]; does the class parameterize those types?");
    }
    addConverter(new ConverterAdapter(converter, typeInfo[0], typeInfo[1]));
}
```

首先根据 converter 获取 ResolvableType，然后将其与 converter 封装成一个 ConverterAdapter 实例，最后调用 addConverter()。ResolvableType 用于封装 Java 的类型。ConverterAdapter 则是 Converter 的一个适配器，它实现了 GenericConverter 和 ConditionalConverter 两个类型转换器。

addConverter() 如下：

```
public void addConverter(GenericConverter converter) {
    this.converters.add(converter);
    invalidateCache();
}
```

直接调用内部类 Converters 的 add() 方法，如下：



```

public void add(GenericConverter converter) {
    Set<ConvertiblePair> convertibleTypes = converter.getConvertibleTypes();
    if (convertibleTypes == null) {
        Assert.state(converter instanceof ConditionalConverter,
            "Only conditional converters may return null convertible types");
        this.globalConverters.add(converter);
    }
    else {
        for (ConvertiblePair convertiblePair : convertibleTypes) {
            ConvertersForPair convertersForPair = getMatchableConverters(convertiblePair);
            convertersForPair.add(converter);
        }
    }
}

```



首先调用 `getConvertibleTypes()` 获取 `ConvertiblePair` 集合，如果为空，则加入到 `globalConverters` 集合中，否则通过迭代的方式依次添加。`ConvertiblePair` 为 `source-to-targer` 的持有者，它持有 `source` 和 `target` 的 `class` 类型，如下：

```

final class ConvertiblePair {
    private final Class<?> sourceType;
    private final Class<?> targetType;

    // 其他代码
}

```

在迭代过程中会根据 `ConvertiblePair` 获取相应的 `ConvertersForPair`，然后 `converter` 转换器加入其中，`ConvertiblePair` 用于管理使用特定 `GenericConverter.ConvertiblePair` 注册的转换器。如下：

```

private static class ConvertersForPair {

    private final LinkedList<GenericConverter> converters = new LinkedList<>();

    public void add(GenericConverter converter) {
        this.converters.addFirst(converter);
    }

    @Nullable
    public GenericConverter getConverter(TypeDescriptor sourceType, TypeDescriptor targetType) {
        for (GenericConverter converter : this.converters) {
            if (!(converter instanceof ConditionalGenericConverter) ||
                ((ConditionalGenericConverter) converter).matches(sourceType, targetType)) {
                return converter;
            }
        }
        return null;
    }
}

```

其实内部就是维护一个 LinkedList 集合。他内部有两个方法：add() 和 getConverter()，实现较为简单，这里就不多介绍了。



## DefaultConversionService

DefaultConversionService 是 ConversionService 的默认实现，它继承 GenericConversionService，GenericConversionService 主要用于转换器的注册和调用，DefaultConversionService 则是为 ConversionService 体系提供一些默认的转换器。在 DefaultConversionService 构造方法中就会添加默认的 Converter，如下：

```
public DefaultConversionService() {
    addDefaultConverters(this);
}

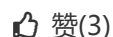
public static void addDefaultConverters(ConverterRegistry converterRegistry) {
    addScalarConverters(converterRegistry);
    addCollectionConverters(converterRegistry);

    converterRegistry.addConverter(new ByteBufferConverter((ConversionService) converterRegistry));
    if (jsr310Available) {
        Jsr310ConverterRegistrar.registerJsr310Converters(converterRegistry);
    }

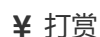
    converterRegistry.addConverter(new ObjectToObjectConverter());
    converterRegistry.addConverter(new IdToEntityConverter((ConversionService) converterRegistry));
    converterRegistry.addConverter(new FallbackObjectToStringConverter());
    if (javaUtilOptionalClassAvailable) {
        converterRegistry.addConverter(new ObjectToOptionalConverter((ConversionService) converterRegistry));
    }
}
```

当然它还提供了一些其他的方法如 addCollectionConverters()、addScalarConverters() 用于注册其他类型的转换器。

至此，从 bean 属性的转换，到 Spring ConversionService 体系的转换器 Converter 以及转换器的管理都介绍完毕了，下篇我们将分析如何利用 ConversionService 实现自定义类型转换器。



赞(3)



打赏

【公告】版权声明 ([http://cmsblogs.com/?page\\_id=1908](http://cmsblogs.com/?page_id=1908))

标签： Spring 源码解析 (<http://cmsblogs.com/?tag=spring-%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90>)

死磕Java (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java>)

死磕Spring (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring>)



上一篇

[Java 8 容器源码文集 \(http://cmsblogs.com/?p=3980\)](http://cmsblogs.com/?p=3980)

下一篇

[【死磕 Spring】—— IOC 之自定义类型转换器 \(http://cmsblogs.com/?p=3985\)](http://cmsblogs.com/?p=3985)

- [【死磕 Redis】—— 如何排查 Redis 中的慢查询 \(http://cmsblogs.com/?p=18352\)](http://cmsblogs.com/?p=18352)
- [【死磕 Redis】—— 发布与订阅 \(http://cmsblogs.com/?p=18348\)](http://cmsblogs.com/?p=18348)
- [【死磕 Redis】—— 布隆过滤器 \(http://cmsblogs.com/?p=18346\)](http://cmsblogs.com/?p=18346)
- [【死磕 Redis】—— 理解 pipeline 管道 \(http://cmsblogs.com/?p=18344\)](http://cmsblogs.com/?p=18344)
- [【死磕 Redis】—— 事务 \(http://cmsblogs.com/?p=18340\)](http://cmsblogs.com/?p=18340)
- [【死磕 Redis】—— Redis 的线程模型 \(http://cmsblogs.com/?p=18337\)](http://cmsblogs.com/?p=18337)
- [【死磕 Redis】—— Redis 通信协议 RESP \(http://cmsblogs.com/?p=18334\)](http://cmsblogs.com/?p=18334)
- [【死磕 Redis】—— 开篇 \(http://cmsblogs.com/?p=18332\)](http://cmsblogs.com/?p=18332)
- [spring boot 源码解析11-ConfigurationClassPostProcessor类加载解析 \(http://cmsblogs.com/?p=9522\)](http://cmsblogs.com/?p=9522)
- [spring boot 源码解析12-servlet容器的建立 \(http://cmsblogs.com/?p=9520\)](http://cmsblogs.com/?p=9520)
- [spring boot 源码解析13-@ConfigurationProperties是如何生效的 \(http://cmsblogs.com/?p=9518\)](http://cmsblogs.com/?p=9518)
- [spring boot 源码解析15-spring mvc零配置 \(http://cmsblogs.com/?p=9516\)](http://cmsblogs.com/?p=9516)
- [spring boot 源码解析16-spring boot外置tomcat部署揭秘 \(http://cmsblogs.com/?p=9514\)](http://cmsblogs.com/?p=9514)
- [spring boot 源码解析17-mvc自动化配置揭秘 \(http://cmsblogs.com/?p=9512\)](http://cmsblogs.com/?p=9512)
- [spring boot 源码解析18-WebMvcAutoConfiguration自动化配置揭秘 \(http://cmsblogs.com/?p=9510\)](http://cmsblogs.com/?p=9510)

