第 3 章　文件处理框架实践

文件扫描器，是用于处理特定文件的一套扫描系统，当容器加载时，优先启动加载，另外提供了路径的过滤、配置的反序列化等附加功能。

在实际项目开发过程中，往往会产生各种类型的文件来存储各种配置或信息，需要通过编程的方式读取这些文件内容。而模块化在最近几年非常热，毕竟只有实现模块化才可以更好地实现资源高内聚，便于进行开发、测试和发布，但是随着应用的模块化，也会导致各种配置文件分散在不同的模块、不同的 Jar 包中，从而大大增加了处理这些资源的难度。

为了解决这个问题，我们构建了文件处理框架，体系性地处理好这些问题。

3.1　概　　述

文件处理框架的设计目标是把文件的扫描、文件的变化、文件的遍历等与文件的实际处理分离，开发者无须关心要处理的文件的具体位置，只要编写文件处理相关的代码即可。文件处理框架会对应用资源进行扫描，然后把扫描到的文件分到开发者开发的文件处理器并由其处理，从而达到对文件进行处理的目的。文件扫描器实现类如图 3-1 所示。

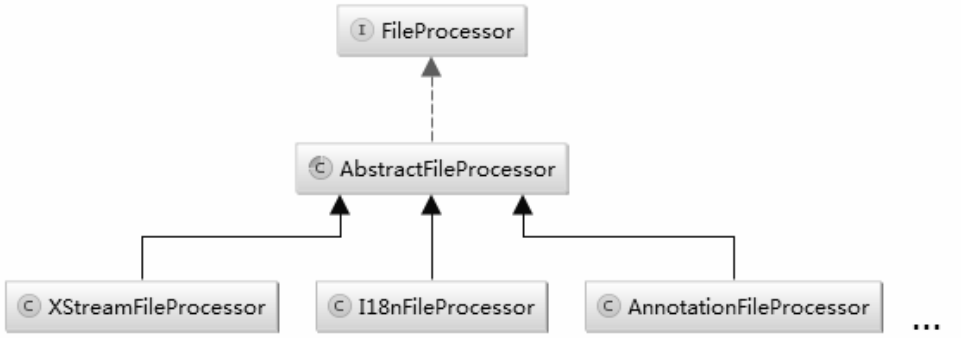

图 3-1　FileProcessor 实现类

3.1.1　FileProcessor 接口

FileProcessor 是文件扫描器接口，开发者通过实现该接口的 isMatch 方法来告诉管理器

哪些文件是自己关心的，对其管理器分发过来的文件进行处理。每种需要处理的文件都需要实现一个 FileProcessor 实现类，接口定义如图 3-2 所示。

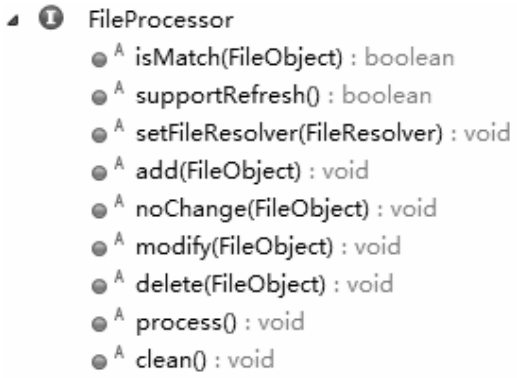

图 3-2　FileProcessor 接口

FileProcessor 接口各方法说明如表 3-1 所示。

表 3-1　FileProcessor方法说明

方　法	说　明
isMach	判断文件是否满足当前 FileProcessor 的匹配规则，若该方法返回为 true，则认为该文件符合规则
supportRefresh	该处理器是否支持文件刷新
setFileResolver	设置 FileResolver，该方法由 FileResolver 调用，以注入一个 FileResolver 实例，以便在需要时使用
add	添加一个符合规则的新文件
noChange	添加一个符合规则且未发生变化的文件
modify	添加一个符合规则且发生变化的文件
delete	删除一个符合规则的文件
process	进行处理，该方法内由用户自行处理所有符合规则的文件
clean	清除扫描器中的文件信息

框架提供了一个抽象类 AbstractFileProcessor，该类实现了 FileProcessor 方法，一般情况下，开发者只需要继承该类，实现以下两个方法即可：

```
//判断文件是否符合规则，符合规则的文件将被收集后，被 process 方法处理
boolean checkMatch(FileObject fileObject);
//处理扫描到的文件
void process();
```

checkMatch 是扫描器的核心方法，开发者通过该方法可对文件进行判断，如果该文件是目标查找对象，则返回 true，否则返回 false。具体匹配流程如图 3-3 所示。

process 则是处理方法，所有对配置文件的处理都需要在这个方法内完成。

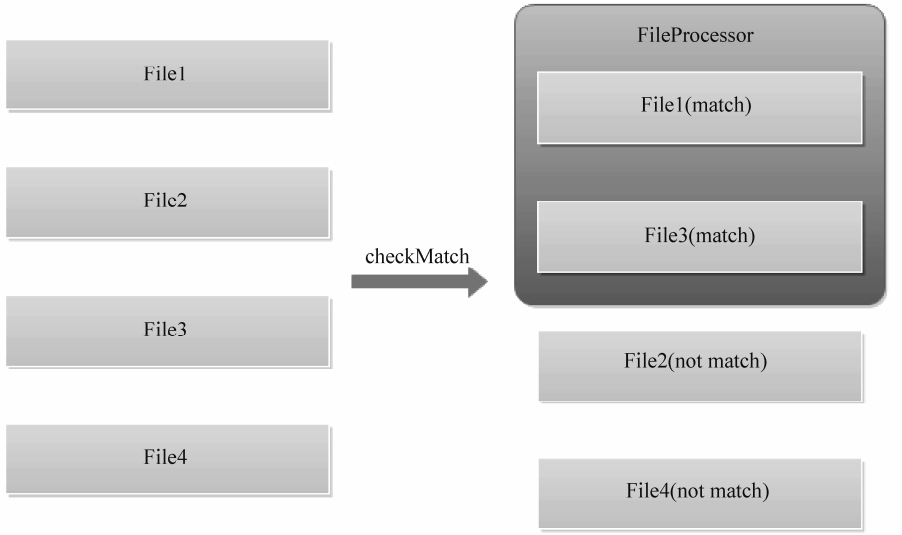

图 3-3　FileProcessor 匹配过程

3.1.2　FileResolver 接口

FileResolver 是资源扫描的主接口，FileProcessor 的实现类被添加至此接口，该实现类会对环境中的资源扫描，系统默认会包含常用的扫描路径，同时也可以由开发者自行添加。路径下的文件会首先用正则表达式判断这些文件是不是要扫描的文件（这一过程主要是为了把一些不需要扫描的文件从扫描范围中去除，加快扫描效率），只有需要扫描的文件，才会被 FileResolver 拿来匹配 FileProcessor 指定的规则，对于符合规则的文件，FileResolver 才会将该文件 add 至 FileProcessor 之中，接口定义如图 3-4 所示。

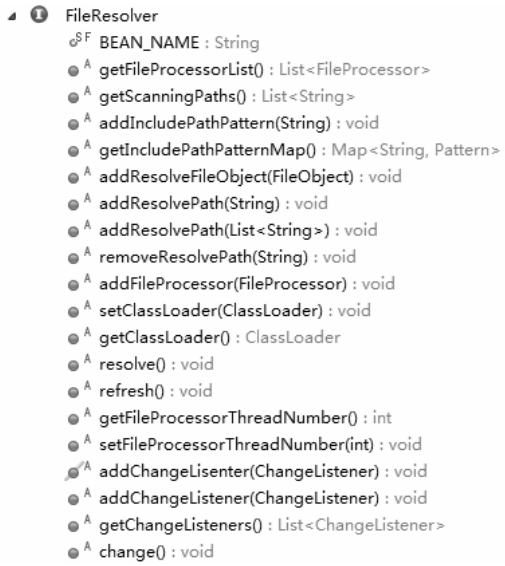

图 3-4　FileResolver 接口

FileResolver 接口各方法说明如表 3-2 所示。

表 3-2　FileResolver方法说明

方　　法	说　　明
getFileProcessorList	获取当前所有的文件处理器
getScanningPaths	获取当前所有扫描路径
addIncludePathPattern	添加需扫描的文件的正则匹配规则
getIncludePathPatternMap	返回扫描 Jar 包的正则匹配规则
addResolveFileObject	添加扫描路径
addResolvePath	添加扫描路径
removeResolvePath	移除扫描路径
addFileProcessor	添加文件处理器
setClassLoader	设置 ClassLoader
getClassLoader	获取 ClassLoader
resolve	开始扫描
refresh	刷新扫描内容
getFileProcessorThreadNumber	获取当前文件处理的线程数目
setFileProcessorThreadNumber	设置当前文件处理的线程数目
addChangeLisenter	添加变更监听器
getChangeListeners	获取变更监听器
change	发起变更，触发变更监听器

框架提供了三个工具方法，用于获取当前工程中需要扫描的文件路径。在框架启动的过程中，这三个路径默认会被添加到框架创建的文件扫描器中。

```
//获取 classPath 中需要扫描的文件
FileResolverUtil.getClassPath(FileResolver resolver);
//获取 web 的 classes 目录下需要扫描的文件
FileResolverUtil.getWebClasses();
//获取 web 的 lib 目录下需要扫描的文件
FileResolverUtil.getWebLibJars(FileResolver resolver);
```

完成扫描路径添加后，即可通过 resolve 发起扫描。

```
fileResolver.addResolvePath(FileResolverUtil.getClassPath(fileResolver)
);
fileResolver.addResolvePath(FileResolverUtil.getWebClasses());
try {
fileResolver.addResolvePath(FileResolverUtil.getWebLibJars(fileResolver
));
} catch (Exception e) {
logger.errorMessage("为文件扫描器添加 webLibJars 时出现异常",e);
}
fileResolver.resolve();
```

FileResolver 提供 addFileProcessor 方法，开发者可以根据需要加入不同的 FileProcessor 实现。FileResolver 通过 resolve 发起扫描时，会扫描所有的注册路径和注册文件中路径符合正则匹配的文件，用已注册的所有 FileProcessor 分别对其进行 checkMatch。如果 match 成功，则会缓存下文件及文件的修改时间。若文件不存在于内存中，则认为是第一次发现该文件，并以新增模式推送给 FileProcessor。如果存在，则判断修改时间是否与内存中已有的修改时间一样，如果不一样，则认为文件发生变化，用最新的文件对象以及修改时间覆盖已有数据。并以修改模式推送至 FileProcessor。如果文件已存在且未修改时间与之前一样，则以无变化模式推送至 FileProcessor。

Resolver 完成后，如有需要（例：文件发生变动、添加了新的配置等情况），开发者可通过 refresh 方法，进行再次刷新扫描。

3.1.3 FileMonitorProcessor 类

FileMonitorProcessor 是基于 FileResolver 的 refresh 方法所做的定时扫描扩展，其核心方法如下。每间隔一段时间，发起 FileResolver 的 refresh 调用，从而发现扫描目录下的文件变化。

```
public void run() {
        while (!stop) {
            try {
                synchronized (synObject) {
                    synObject.wait(interval * MILLISECOND_PER_SECOND);
                    if (!stop) {
                        resolver = BeanContainerFactory.getBeanContainer(
                            this.getClass().getClassLoader()).getBean(
                            "fileResolver");
                        resolver.refresh();//刷新文件列表
                    }
                }
            } catch (InterruptedException e) {
                LOGGER.errorMessage(e.getMessage(), e);
            }
        }
    }
```

3.2 基础文件扫描器

Tiny 框架内置了几个文件扫描器，这些扫描器是 Tiny 的核心，缺一不可，它们负责

Tiny 启动过程中，基础功能的初始化。

3.2.1　XStreamFileProcessor 类

XStreamFileProcessor 是用于处理对象序列化的文件扫描器。采用基于 xstream 的方式作为序列化和反序列化方案，通过 Xstream 注解规范，将 Java 对象的实例快速序列化或者将配置文件中的内容快速反序列化为该对象的实例。

首先定义需要进行反序列化或者序列化的对象，需要通过 xstream 来完成此功能，并需要加上相关的注解，如下所示。

```
@XStreamAlias("components")
public class ComponentDefines {
    @XStreamImplicit
    private List<ComponentDefine> componentDefines;
    //省略 setter/geter
}

@XStreamAlias("component")
public class ComponentDefine implements Serializable{
        @XStreamAsAttribute
        private String category;
        @XStreamAsAttribute
        private String name;
        @XStreamAsAttribute
        private String bean;
        @XStreamAsAttribute
        private String title;
        @XStreamAsAttribute
        private String icon;
        @XStreamAsAttribute
        @XStreamAlias("short-description")
        private String shortDescription;
        @XStreamAsAttribute
        @XStreamAlias("long-description")
        private String longDescription;
        @XStreamImplicit
        private List<Result> results;
        //省略 setter/getter
}

@XStreamAlias("result")
public class Result {
        @XStreamAsAttribute
```

```
        private String name;
        @XStreamAsAttribute
        private String title;
        @XStreamAsAttribute
        private String type;
        @XStreamAsAttribute
        private Boolean array;
        @XStreamAsAttribute
        private Boolean required;
        @XStreamAlias("collection-type")
        @XStreamAsAttribute
        private String collectionType;
        //省略 setter/getter
}
```

XStream 相关注解含义如表 3-3 所示，更多的 XStream 知识请读者自己查找资料学习。

表 3-3 XStream注解说明

注　解	说　明
XStreamAsAttribute	将当前配置项作为节点属性
XStreamAlias	为当前配置项设置别名
XStreamImplicit	不为当前集合节点列表设置专门的父节点

创建一个*.xstream.xml 文件，用于配置需要处理的对象，此处配置如下。package-name 是命名空间，用于区分不同模块的对象定义，避免出现命名冲突。

```
<xstream-configuration package-name="flow">
    <xstream-annotation-classes>
    <xstream-annotation-class
        class-name="org.tinygroup.flow.config.ComponentDefines" />
    </xstream-annotation-classes>
</xstream-configuration>
```

XStreamFileProcessor 通过 FileResolver 扫描器收集*.xstream.xml 文件。

XStreamFileProcessor 会读取所有的*.xstream.xml 文件，根据文件中配置的 class，读取对应的注解配置。相关信息会根据配置的 package-name 存入 XStreamFactory。

```
XStream loadXStream = XStreamFactory.getXStream();
XStreamConfiguration   xstreamConfiguration   =   (XStreamConfiguration)
loadXStream
                .fromXML(fileObject.getInputStream());
XStream xStream = XStreamFactory.getXStream(xstreamConfiguration
                .getPackageName());
loadAnnotationClass(xStream, xstreamConfiguration);//根据 XML 配置加载注释类
```

```
if (xstreamConfiguration.getxStreamClassAliases() != null) {
 processClassAliases(xStream, xstreamConfiguration.getxStreamClassAliases());
}
```

完成以上步骤后，开发者只需要从 XStreamFactory 中获取该 XStream 即可进行对象的正反序列化处理了。

注意：代码中的 flow 必须和对应的*.xstream.xml 里面的命名空间 package-name 保持一致。

```
XStream stream = XStreamFactory.getXStream("flow");
//将 fileObject 文件中的 xml 内容反序列化为 ComponentDefines 对象
ComponentDefines components =
(ComponentDefines) stream.fromXML(fileObject.getInputStream());
```

3.2.2　I18nFileProcessor 类

I18nFileProcessor 是国际化配置文件的扫描器。它收集所有父文件夹是 i18n 的*.properties 或者*.cproperties 文件，将其作为国际化语言文件。工程结构如图 3-5 所示。

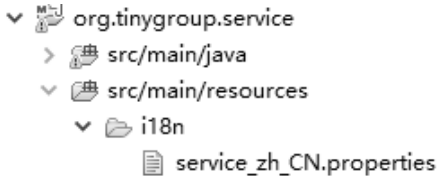

图 3-5　国际化

国际化文件命名规范为“模块名_Locale 信息.后缀名”，如 service_zh_CN.properties。

3.2.3　Annotation 扫描器

Annotation 扫描器由 AnnotationFileProcessor 和 AnnotationClassFileProcessor 组成。前者负责收集注解和注解处理类的映射关系，而后者负责查找符合规则的.class。

框架提供注解解析方案，该方案允许开发者用预先配置的处理器去处理配置了该注解的类，整个处理过程由框架完成，开发者只需要配置注解与处理类的映射关系。

AnnotationFileProcessor 配置文件名格式为*.annotation.xml。

```
<annotation-class-matchers>
    <annotation-class-matcher class-name=".*\.annotation\.Annotation.*"
        annotation-type=".*Test" annotation-id="test">
        <processor-beans>
```

```
            <processor-bean enable="true" name="classAction">
            </processor-bean>
        </processor-beans>
        <annotation-method-matchers>
            <annotation-method-matcher method-name="method.*"
                annotation-type=".*Test">
                <processor-beans>
                    <processor-bean enable="true" name="methodAction">
                    </processor-bean>
                </processor-beans>
            </annotation-method-matcher>
        </annotation-method-matchers>
        <annotation-property-matchers>
            <annotation-property-matcher
                property-name="field.*" annotation-type=".*Test">
                <processor-beans>
                    <processor-bean enable="true" name="propertyAction">
                    </processor-bean>
                </processor-beans>
            </annotation-property-matcher>
        </annotation-property-matchers>
    </annotation-class-matcher>
</annotation-class-matchers>
```

*.annotation.xml 文件格式如上所示，开发者可以为类注解、方法注解和属性注解分别定义不同的处理器。一个注解可以有多个注解处理器类，不同处理类之间相互独立。annotation-id 是注解唯一标识符，只存在于 annotation-class-matcher 节点。当注解被引用时，以该 annotation-id 为引用标识，具体 xml 说明如表 3-4 所示。

表 3-4　注解xml说明

节点/属性	说　　明
annotation-class-matcher	类配置节点
annotation-method-matchers	方法配置节点列表
annotation-method-matcher	方法配置节点
annotation-property-matchers	属性配置节点列表
annotation-property-matcher	属性配置节点
class-name	注解所在类全路径的正则匹配规则，只有当类名匹配该正则时，框架才会去解析该类，判断该类是否有配置相应的注解
annotation-id	类注解处理规则唯一标识
method-name	方法名正则匹配规则
property-name	属性名正则匹配规则
annotation-type	注解类型正则匹配规则

续表

节点/属性	说　　明
processor-beans	注解处理类 bean 配置节点列表
processor-bean	注解处理类 bean 配置节点
enable	是否启用该配置
name	注解处理器 bean 的 name

AnnotationClassFileProcessor 负责查找.class，并进行解析判断.class 中是否存在已定义处理器的注解。如果存在，则将其交给处理器进行处理。

由于实际应用中.class 数量过于巨大，如果对所有的.class 进行扫描，并读取判断是否有配置该注解，势必造成解析进度缓慢。实际上，例如第三方 jar 包等包，根本无须去解析里面的.class，因为开发者自定义的注解只会存在于自己的应用包中。基于以上原因，AnnotationClassFileProcessor 提供了正则匹配规则，只有符合该正则规则的才会去解析对应的注解处理器规则，该配置在 application.xml 中。

```
<annotation-configuration>
    <annotation-mapping id="service" value="(.)*Service"></annotation-mapping>
    <annotation-mapping id="service" value="(.)*Adapter"></annotation-mapping>
    <annotation-mapping id="validate" value="org\.tiny\..*"></annotation-mapping>
</annotation-configuration>
```

上例中配置了.class 解析规则，id 是前文中提到的配置项 annotation-id。只有类名匹配正则“(.)*Service”或者“(.)*Adapter”才会被框架去解析是否存在注解处理器 service 所声明的注解类型。同样，只有类全路径匹配“org\.tiny\..*”才会被框架去解析是否存在 validate 所声明的注解类型。

本小节提供了注解与注解处理类的映射关系配置方案，开发者根据该方案提供的方式，可以为注解及注解处理类成功建立联系。框架得到该配置后，在框架启动时或者 FileResolver 进行 Resolve 时，会查找到符合规则的 java 类的.class，通过该注解处理器类进行处理。整个过程，全部由框架来负责完成，开发者不需要做任何其他工作。而查找符合规则的.class 的任务则由 annotationClassFileProcessor 来完成。

3.2.4　SpringBeansFileProcessor 类

SpringBeansFileProcessor 是用于扫描框架中 Spring beans 配置文件的扫描器。框架中使用了 Spring beans 容器作为对象容器。开发者需要遵循框架定义的 beans 文件命名规则，以*.springbeans.xml 或者*.beans.xml 作为 Spring beans 配置文件的名称，在配置文件格式上，完全依照 Spring 的规范，不存在任何差异。

3.3 完 整 示 例

文件处理的应用场景可能不同，我们就区分开来写两个示例，分别是文件处理框架单独使用和 Tiny 框架集成文件处理框架，详细的示例可以参考示例工程。

本章应用示例工程代码，请参考附录 A 中的 org.tinygroup.fileresolver.demo（文件处理框架实践示例工程）。

3.3.1 单独使用

开发者可直接创建 FileResolver 添加各种配置，然后调用 resolve 方法进行文件扫描。

```
FileResolver fileResolver = new FileResolverImpl();
fileResolver.addResolvePath(path);                    //path 为需要扫描的路径
fileResolver.addIncludePathPattern(TINY_JAR_PATTERN);//路径匹配正则
fileResolver.addFileProcessor(new SpringBeansFileProcessor());
                                                      //spring 配置文件扫描器
fileResolver.addFileProcessor(new ConfigurationFileProcessor());
                                                      //组件配置扫描器
fileResolver.resolve();//开始扫描
```

3.3.2 通过配置文件配置

在本书提供的源代码中，还有另一种方式进行配置，即直接在配置文件中进行配置。

首先，在 application.xml 中配置一个启动器。

```
<application-processors>
<application-processor
bean="fileResolverProcessor"></application-processor>
</application-processors>
```

在 Application.springbeans.xml 中通过 Bean 注入的方式来为 FileResolver 注入各种 FileProcessor。开发者只需要将自己开发的文件扫描器的 bean 配置到 fileProcessorList 属性之下即可。

```
 <bean id="fileResolver" scope="singleton"
   class="org.tinygroup.fileresolver.impl.FileResolverImpl">
   <property name="fileProcessorList">
```

```
        <list>
            <ref bean="i18nFileProcessor" />
            <ref bean="xStreamFileProcessor" />
            <ref bean="xmlServiceFileProcessor" />
            <ref bean="tinyFilterFileProcessor" />
            <ref bean="tinyProcessorFileProcessor" />
            <ref bean="fullContextFileFinder" />
        </list>
    </property>
</bean>
```

开发者还可以在 application.xml 中对扫描路径、匹配规则等进行配置。

```
<file-resolver-configuration
       resolve-classpath="true">
   <class-paths>
       <!– 需要扫描的应用外部路径，通过 addResolvePath 添加到扫描器-->
       <!-- <class-path path="{TINY_WebROOT}\Web-INF\lib" /> -->
   </class-paths>
   <include-patterns>
        <!– 需扫描的文件的正则匹配规则,通过 addIncludePathPattern 添加到扫描器-->
         <include-pattern pattern="org\.tinygroup\.(.)*\.jar"/>
       </include-patterns>
   </file-resolver-configuration>
```

另外，还可以在 application.xml 中配置定时扫描的刷新功能，相关配置项如下。Interval 为每次刷新的间隔时间，enable 表示是否启用定时刷新功能。该功能开启后，每过指定间隔时间，都会调用 FileResolver 的 refresh 方法。

```
<file-monitor interval="10" enable="false" />
```

3.4　本 章 总 结

本章讲解了框架的文件扫描体系，框架通过此种方式实现了自发现式配置。开发者不再需要刻意将项目内归属于不同包或者工程的配置集中放置了，只需要实现文件扫描器接口 FileProcessor，即可将离散分布在各处的配置文件收集并进行统一处理。

通过本章提供的技术，可以方便地处理各种类型的资源文件，而不用关心怎么查找它、怎么遍历它以及怎么监视它的变化等等，程序员只要关心要处理什么文件和怎么处理文件两个必要的步骤上，简化了开发环节，降低了开发工作量。

同时由于扫描文件及监控文件的变化由框架进行，也避免了不同的文件处理逻辑多次

扫描整个运行环境而导致的大量性能损耗，使得同样的事情只做一次成为可能。

本章还简要介绍了框架中基于自发现机制提供的文件序列化方案。采用 xstream 进行文件序列化和反序列化处理。该方式需要首先在 Java 对象上配置 xstream 相关的注解，再将该类配置为*.stream.xml。使用时，通过该 xstream 实例即可直接将文件反序列化 Java 对象。感兴趣的读者可以直接查看 Tiny 框架源码中的相关内容。