

Tomcat 源代码调试 - 看不见的 Shell 第二式之隐藏任意 Jsp 文件

n1nty n1nty 2017-06-29 09:50

作者原创，转载请注明版权。

这篇笔记我尽量少贴代码，有兴趣的可以自己去看一下。

需要知道的背景知识：

1. 在 tomcat 的 conf/web.xml 文件中配置了一个如下的 servlet：

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  <init-param>
    <param-name>fork</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>xpoweredBy</param-name>
    <param-value>>false</param-value>
  </init-param>
  <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

这意味着，tomcat 接收到的所有的 jsp 或jspx 的请求，都会转交给 org.apache.jasper.servlet.JspServlet 来处理，由它来将请求导向至最终的位置。

2. Jsp 文件会被转换为 Java 文件，并随后被编译为 class。转换后的文件与编译后的 class 默认保存在 Tomcat 下的 work 目录中。请求最终会被导向至从 Jsp 文件编译出来的 class 的对象上。

3. Jsp 被编译并被加载实例化后，会被封装在一个 JspServletWrapper 对象中。在 Tomcat 中，每一个 Context 都对应有一个 JspRuntimeContext 对象，该对象中以 Map 的形式，以 path（如 /index.jsp）为 key 保存了当前 Context 中所有的 JspServletWrapper 对象。

4. 被编译并且被 Tomcat 加载后（创建了对应的 JspServletWrapper 对象后），Jsp 文件以及转换出来的 Java 文件以及由 Java 文件编译出来的 class 文件，在一定程度上来说，都是可有可无的。

这里简述一下 Tomcat 会在什么时候对 Jsp 进行编译：

1. 当 Tomcat 处于 development 模式时（这是默认的），当一个 Jsp 第一次被请求时，会对被请求的文件进行编译。随后，每次请求时，都会对文件进行更新检查，一旦发现源 Jsp 文件有变更，则将重新编译。而如果发现源 Jsp 文件不存在了，则会出现 404，这是我们要“欺骗”的一个地方。
2. 当 Tomcat 处于非 development 模式，且 JspServlet 的初始化参数 checkInterval 的值大于 0 的时候，Tomcat 将采用后台编译的方式。这种情况下，当一个 Jsp 第一次被访问的时候，它将会被编译。随后每隔指定的时间，会有一个后台线程对这些 Jsp 文件进行更新检查，如果发现文件有更新，则将在后台进行重新编译，如果发现文件不存在了，将从 JspRuntimeContext 中删除对应的 JspServletWrapper 对象，导致我们随后的访问出现 404。这是我们要欺骗的另一个地方，虽然看起来与上面是一样的，但是体现在代码中却不太一样。

讲到这里，所谓“隐藏任意 Jsp 文件”的原理也就很简单了。只要在 Jsp 编译完成后，删掉原有 Jsp 文件，并“欺骗”Tomcat 让它认为文件依然存在，就可以了。

简述一下 Tomcat 接收请求的过程，当然这里只简述请求到达 JspServlet 后发生的事情，之前的事情就太多了。这里从 JspServlet 的 serviceJspFile 开始说起，代码如下：

```
private void serviceJspFile(HttpServletRequest request,
                            HttpServletResponse response, String jspUri,
                            boolean precompile)
    throws ServletException, IOException {

    JspServletWrapper wrapper = rctx.getWrapper(jspUri);
    if (wrapper == null) {
        synchronized(this) {
            wrapper = rctx.getWrapper(jspUri);
            if (wrapper == null) {
                // Check if the requested JSP page exists, to avoid
                // creating unnecessary directories and files.
                if (null == context.getResource(jspUri)) {
                    handleMissingResource(request, response, jspUri);
                    return;
                }
                wrapper = new JspServletWrapper(config, options, jspUri,
                                                  rctx);
                rctx.addWrapper(jspUri, wrapper);
            }
        }
    }
}
```

```
    try {
        wrapper.service(request, response, precompile);
    } catch (FileNotFoundException fnfe) {
        handleMissingResource(request, response, jspUri);
    }
}
```

它的主要作用就是检查 JspRuntimeContext 中是否已经存在与当前 Jsp 文件相对应的 JspServletWrapper（如果存在的话，说明这个文件之前已经被访问过了）。有的话就取出来，没有则检查对应的 Jsp 文件是否存在，如果存在的话就新建一个 JspServletWrapper 并添加到 JspRuntimeContext 中去。

随后会进入 JspServletWrapper 的 service 方法，如下（我对代码进行了删减，只看与主题有关的部分）：

```
public void service(HttpServletRequest request,
                    HttpServletResponse response,
                    boolean precompile)
    throws ServletException, IOException, FileNotFoundException {

    Servlet servlet;

    try {

        /*
         * (1) Compile
         */
        if (options.getDevelopment() || firstTime ) {
            synchronized (this) {
                firstTime = false;

                // The following sets reload to true, if necessary
                ctxt.compile();
            }
        } else {
            if (compileException != null) {
                // Throw cached compilation exception
                throw compileException;
            }
        }

        /*
         * (2) (Re)load servlet class file
         */
        servlet = getServlet();

        // If a page is to be precompiled only, return.
        if (precompile) {
            return;
        }
    }
```

```

    } catch (ServletException ex) {
        .....
    }

    /*
     * (4) Service request
     */
    if (servlet instanceof SingleThreadModel) {
        // sync on the wrapper so that the freshness
        // of the page is determined right before servicing
        synchronized (this) {
            servlet.service(request, response);
        }
    } else {
        servlet.service(request, response);
    }
} catch (UnavailableException ex) {
    ....
}
}

```

可以看到，主要流程就是编译 Jsp，然后进入编译出来的 Jsp 的 service 方法，开始执行 Jsp 内的代码。这里先判断当前 Jsp 是不是第一次被访问，或者 Tomcat 是否处于 development 模式中。如果是，则会进入

```
ctxt.compile();
```

对 Jsp 进行编译。ctxt 是 JspCompilationContext 的对象，该对象内封装了与编译 Jsp 相关的所有信息，每一个 JspServletWrapper 里面都有一个自己的 JspCompilationContext。也就是在 compile 方法里面，对 Jsp 文件的更改以及删除做了检查。

而当 Tomcat 利用后台线程来对 Jsp 的更新删除做检查的时候，是不会经过这里的，而是直接进入 JspCompilationContext 的 compile 方法（也就是上文的 ctxt.compile() 方法）。代码如下：

```

public void compile() throws JasperException, FileNotFoundException {
    createCompiler();
    if (jspCompiler.isOutDated()) {
        if (isRemoved()) {
            throw new FileNotFoundException(jspUri);
        }
        try {
            jspCompiler.removeGeneratedFiles();
            jspLoader = null;
            jspCompiler.compile();
            jsw.setReload(true);
            jsw.setCompilationException(null);
        } catch (JasperException ex) {
            // Cache compilation exception
            jsw.setCompilationException(ex);
            if (options.getDevelopment() && options.getRecompileOnFail()) {

```

```

        // Force a recompilation attempt on next access
        jsw.setLastModificationTest(-1);
    }
    throw ex;
} catch (FileNotFoundException fnfe) {
    // Re-throw to let caller handle this - will result in a 404
    throw fnfe;
} catch (Exception ex) {
    JasperException je = new JasperException(
        Localizer.getMessage("jsp.error.unable.compile"),
        ex);
    // Cache compilation exception
    jsw.setCompilationException(je);
    throw je;
}
}
}

```

JspCompilationContext 对象内有一个 Compile 对象，用它来对 Jsp 进行更新检查以及编译。jspCompile.isOutDated 方法代码如下：

```

public boolean isOutDated(boolean checkClass) {

    if (jsw != null
        && (ctxt.getOptions().getModificationTestInterval() > 0)) {

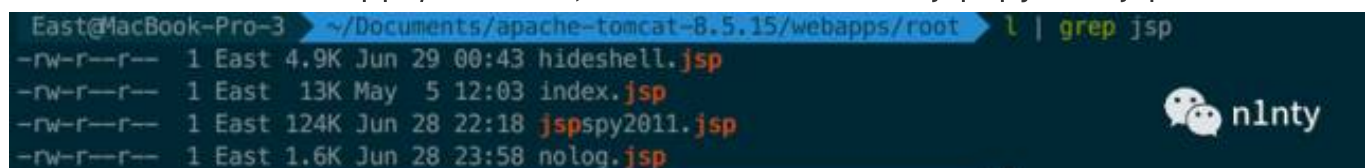
        if (jsw.getLastModificationTest()
            + (ctxt.getOptions().getModificationTestInterval() * 1000) > System
                .currentTimeMillis()) {
            return false;
        }
        jsw.setLastModificationTest(System.currentTimeMillis());
    }

    Long jspRealLastModified = ctxt.getLastModified(ctxt.getJspFile());
    if (jspRealLastModified.longValue() < 0) {
        // Something went wrong - assume modification
        return true;
    }
    .....
}

```

我们只需要让此方法返回 false，那么无论 Tomcat 在何时对 Jsp 文件进行编译或者更新检查，都会认为这个 JspServletWrapper 对象的 Jsp 文件没有发生任何更改，所以也就不会发现文件被删掉了。它会继续保留这个 JspServletWrapper 对象以供客户端访问。

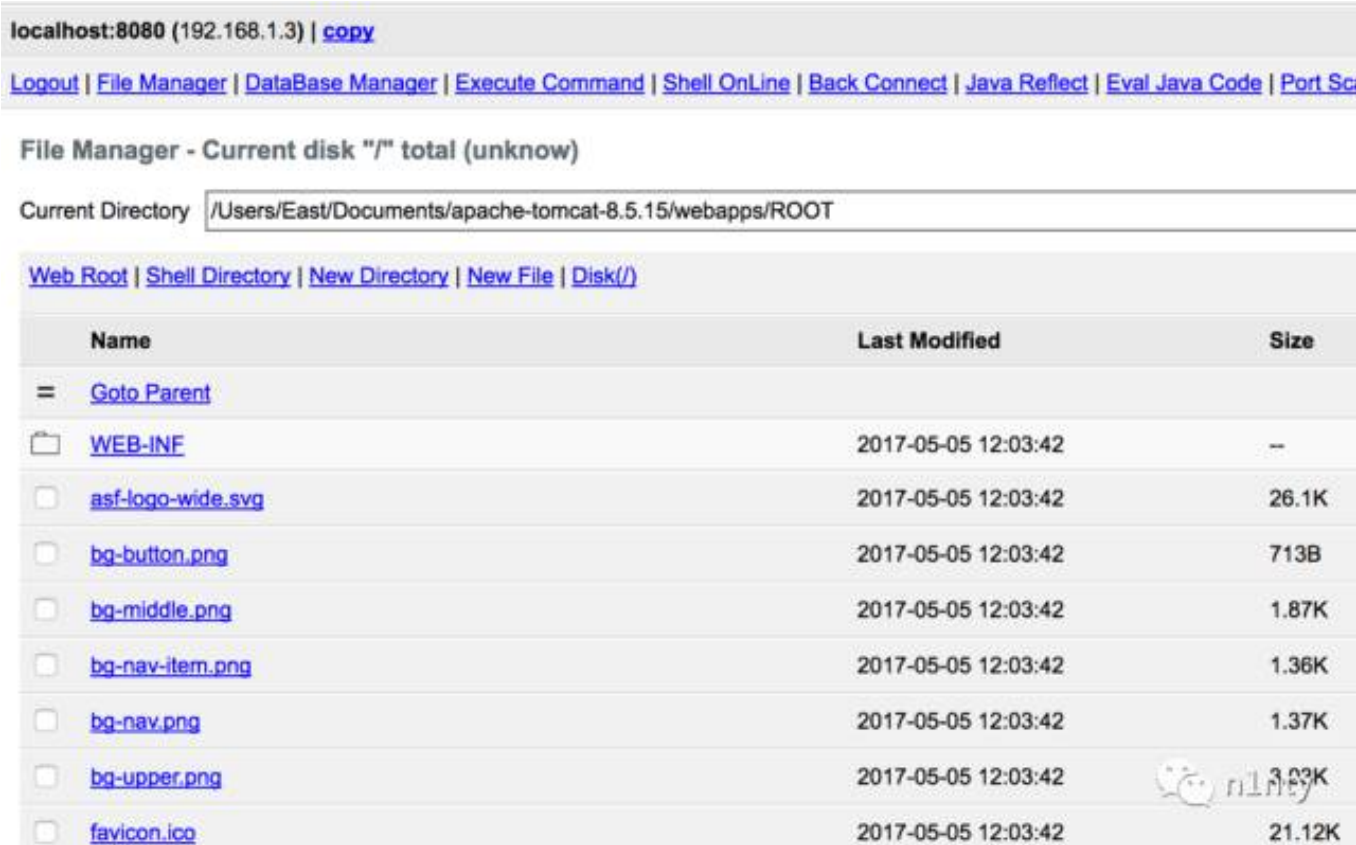
后面就没有什么好说的了，如何进行“欺骗”，大家直接看效果吧。将 hideshow.jsp（在后面提供）放在 webapps/ROOT 下，同目录下有传说中的 jspspy2011.jsp：



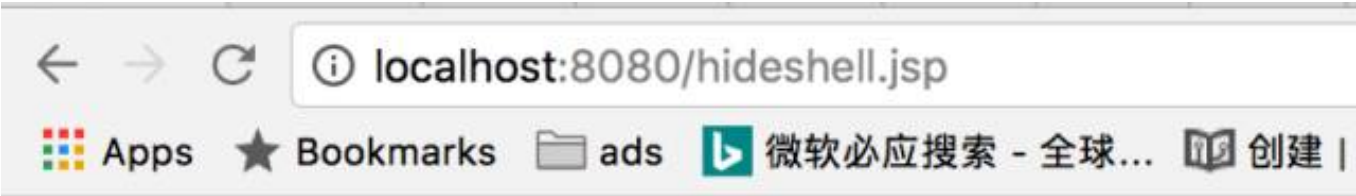
```

East@MacBook-Pro-3 ~/Documents/apache-tomcat-8.5.15/webapps/root l | grep jsp
-rw-r--r--  1 East  4.9K Jun 29 00:43 hideshow.jsp
-rw-r--r--  1 East  13K May  5 12:03 index.jsp
-rw-r--r--  1 East 124K Jun 28 22:18 jspspy2011.jsp
-rw-r--r--  1 East  1.6K Jun 28 23:58 nolog.jsp

```

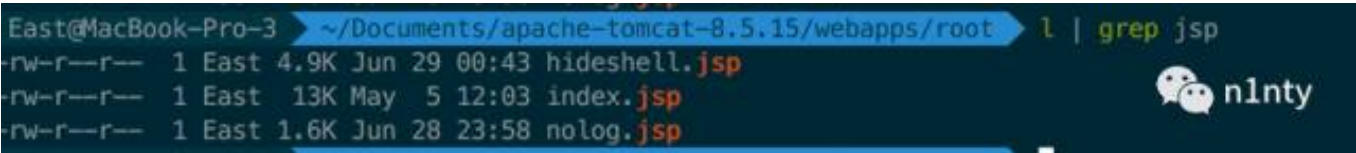


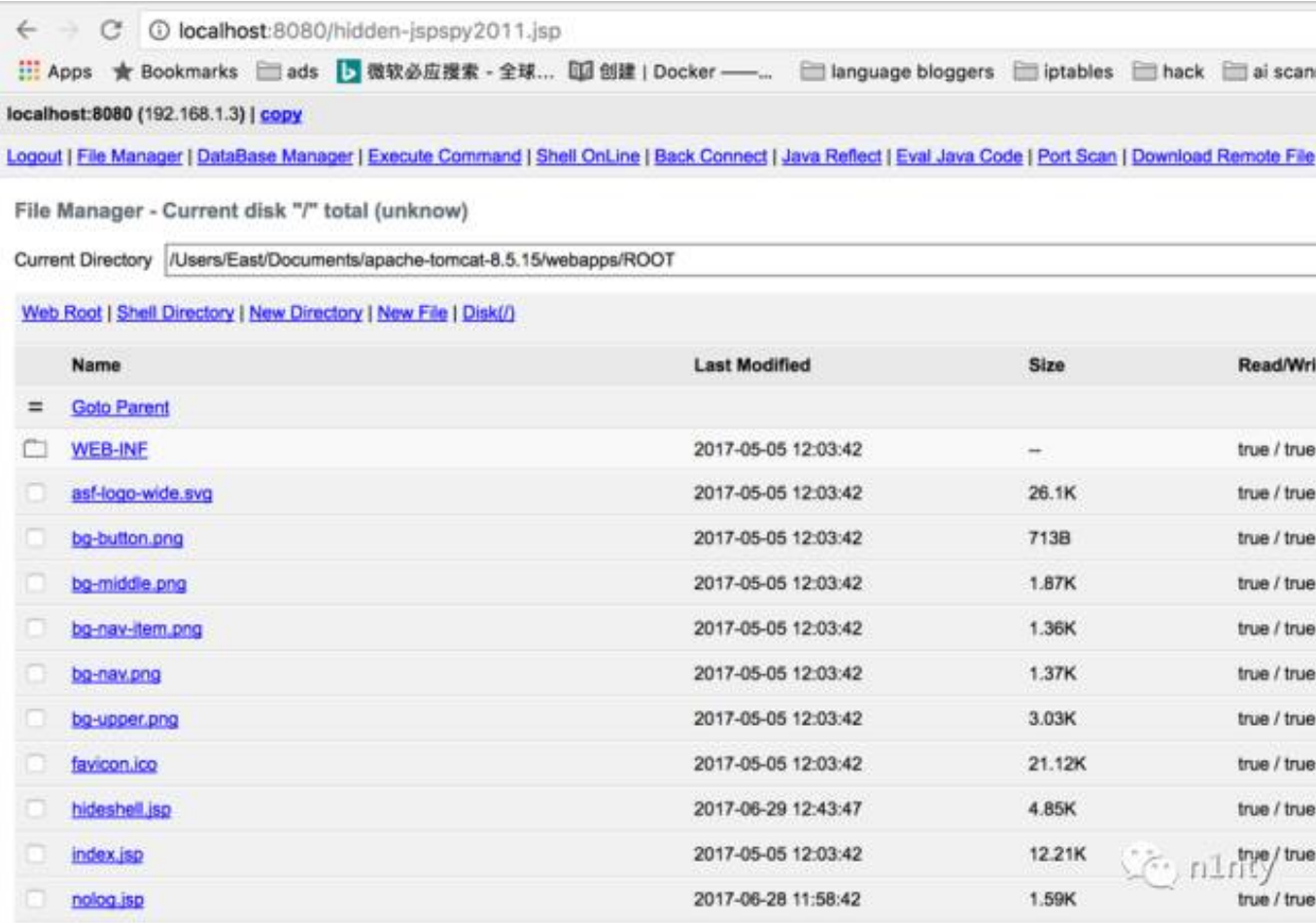
Tomcat 启动后，先访问一下 `jspspy2011.jsp`，目的是为了让 Tomcat 将它编译，并生成 `JspServletWrapper` 保存在 `JspRuntimeContext` 中（其实我们也可以自己用代码来编译，但是我太懒）。然后再访问 `hideshell.jsp`，如下图：



- [Hide /hideshell.jsp](#)
- [Hide /jspspy2011.jsp](#)

点击 "Hide `/jspspy2011.jsp`"，会发现 `webapps/ROOT` 目录下的 `jspspy2011.jsp` 消失了，再访问 `jspspy2011.jsp` 出现了 404。Shell 被“隐藏”了，而且访问路径被更改成了 `hidden-jspspy2011.jsp`：





同时再回到 hideshow.jsp，它会提示 /hidden-jspspy2011.jsp 是一个疑似的隐藏文件：



hideshow.jsp 会尝试将被隐藏的 Jsp 文件与它生成的 Java 与 class 文件全部删掉。但是我发现如果 Jsp 中使用了内部类，这些内部类所编译出来的 class 不会被删掉。

“隐藏”任意 Jsp 文件到此已经实现了。可是虽然文件看不到了，当我们在访问隐藏后的路径的时候，依然会产生日志。那么下一篇笔记，有可能分享一下在隐藏 Shell 的同时，如何隐藏掉它们产生的访问日志。