Tomcat

权威指南

极客学院出版

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

前言

Tomcat 是由 Apache 软件基金会下属的 Jakarta 项目开发的一个 Servlet 容器，按照 Sun Microsystems 提供的技术规范开发出来，Tomcat 8 实现了对 Servlet 3.1 和 JavaServer Page 2.3（JSP）的支持，并提供了作为 Web 服务器的一些特有功能，如 Tomcat 管理和控制平台、安全域管理和 Tomcat 附加组件等。

适用人群

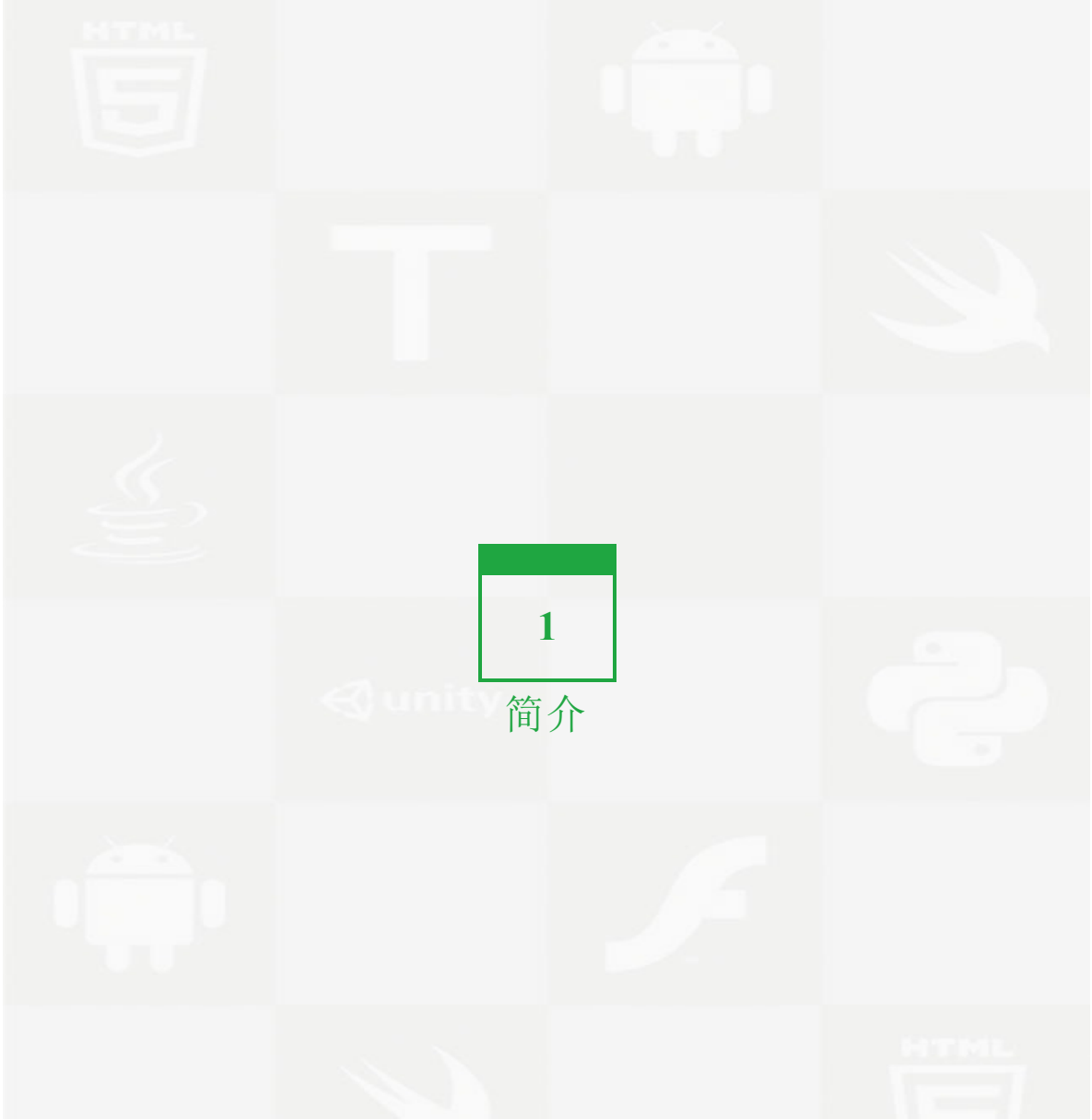
对管理员和 Web 站点管理员而言，具有较强的参考价值；对于开发或产品中要使用 Tomcat 作为 Web 应用程序服务器的开发者而言，这是一本有用的教程。

学习前提

本教程包含了 Tomcat 的基础功能，也有高级功能。对于初学者，你最好按照教程一步步搭建环境，这样才能保证你更好的理解 Tomcat 的高级功能。

原文出处：http://tomcat.apache.org/tomcat-8.0-doc/introduction.html

更新日期	更新内容
2015-06-18	Tomact 教程



1 简介

对于系统管理员以及 Web 开发者来说，在开始学习 Tomcat 之前应该熟悉一些重要内容。本章概述了 Tomcat 容器背后的一些概念和术语，以及你可能需要的一些帮助资源。

术语

阅读这些文档时，会碰到一些术语。其中一些是 Tomcat 的专有术语，另一些则是由 Servlet 与 JSP 规范所定义的术语。

- **Context** 简单说，上下文就是指 Web 应用程序。
- **Term2** 术语 2
- **Term3** 术语 3

目录与文件

贯穿所有文档，你将会注意到很多地方都提到了 **$CATALINA_HOME**。这是 Tomcat 安装的根目录。假如文档中某处出现“该信息应该位于 `$CATALINA_HOME/README.txt` 文件中”，那它其实是指在 Tomcat 安装根目录下查看 README.txt 文件。另外，还可以配置多个 Tomcat 实例，只需为每一个实例都定义一个 **$CATALINA_BASE** 即可。当然，如果没有配置多个实例，那么 **$CATALINA_BASE** 其实就相当于 **$CATALINA_HOME**。

以下是 Tomcat 的一些关键目录：

- **/bin** 存放用于启动及关闭的文件，以及其他一些脚本。其中，UNIX 系统专用的 `*.sh` 文件在功能上等同于 Windows 系统专用的 `*.bat` 文件。因为 Win32 的命令行缺乏某些功能，所以又额外地加入了一些文件。
- **/conf** 配置文件及相关的 DTD。其中最重要的文件是 server.xml，这是容器的主配置文件。
- **/log** 日志文件的默认目录。
- **/webapps** 存放 Web 应用的相关文件。

配置 **Tomcat**

本部分内容将带你熟悉容器配置过程中用到的基本信息。

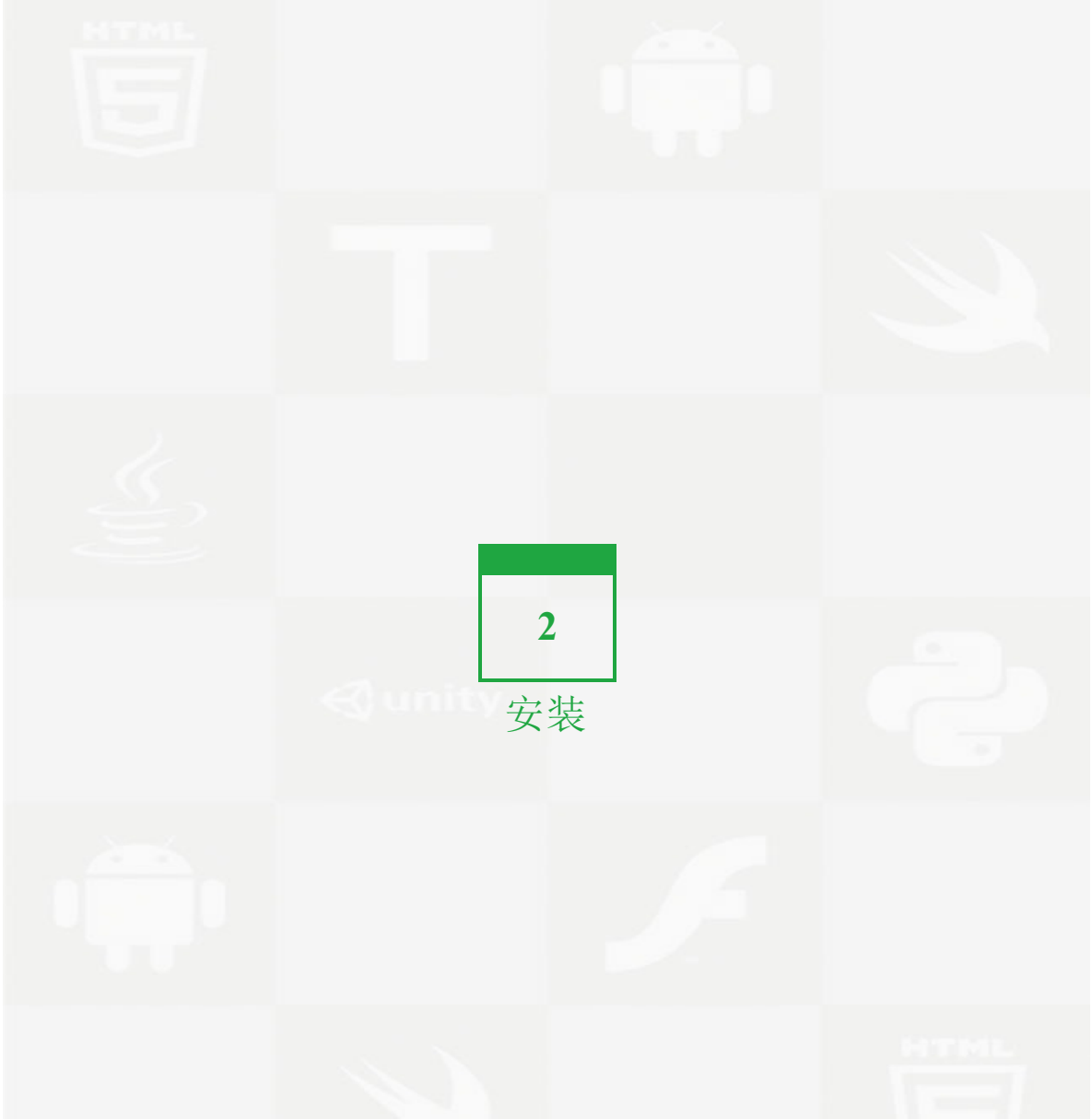
配置文件中的所有信息在启动时才被读取，所以改动文件后，必须重启容器才能使之生效。

帮助

尽管我们努力使文档变得清晰，易于理解，但可能还会有遗漏之处，所以假设遇到不解之处，你可以参考下面这些网站和邮件列表。

注意，根据 Tomcat 主要版本的不同，有些问题和解决方案也存在差异，所以网上的一些文档可能并不是针对 Tomcat 8，而是早前的一些版本。

- 当前文档 多数文档会列出一些可能性。一定要完整地读完相关的文档，这能帮你省下很多时间和精力，因为有可能努力去网上搜寻的答案可能一直就在我们旁边呆着。
- **Tomcat FAQ**
- **Tomcat WIKI**
- **jGuru** 上的 Tomcat FAQ
- **Tomcat** 邮件列表归档 很多网站都保存着 Tomcat 的一些邮件列表。因为有些链接会随着时间发生变化，所以按一定条件去 Google 上搜索。
- Tomcat用户邮件列表。可以点击此处订阅。如果你发现自己的问题无人回应，那么恭喜你，问题可能已经在邮件列表归档或者某个 FAQ 里解答过了。一般来说，有关 Web 应用开发的问题时常会被人提问并随即得到解答，但还是请把问题范围限定在 Tomcat 领域内。
- Tomcat 开发者邮件列表。可以点击此处订阅。该邮件列表是针对 Tomcat 自身的开发研讨而专门设立的。Tomcat 配置问题以及开发及运行应用时遇到的问题更适于在 Tomcat 用户邮件列表中提问。



2

安装

本章概述

可利用多种方法把 Tomcat 安装到不同的平台上。关于 Tomcat 安装方面的重要文档是 RUNNING.txt。如果本节内容尚未能解决你的某些困惑，建议查阅该文档获取帮助。

Windows 系统下的安装

利用 Windows 安装程序可以轻松地在 Windows 系统下安装 Tomcat。无论是在界面还是在功能上，Windows 安装程序都有向导式安装程序，只需在以下几个方面稍加注意：

- 以 **Windows** 服务的形式进行安装 利用多种配置，Tomcat 可以安装为 Windows 服务。在组件页面勾选复选框，将服务设置为“自动”启动，这样当 Windows 启动时，Tomcat 也随即启动。为了获取最佳的安全性，可以把该服务作为单独用户来运行，并降低权限（详情参看 Windows 服务管理工具及其相关文档）
- **Java** 位置 为了运行服务，安装程序通常会提供默认的 JRE。安装程序使用注册表来确认 JRE 的基础路径，这可能是 Java 7 或 更新的版本，还可能包括安装在完整 JDK 中作为其一个部分存在的 JRE。在 64 位操作系统下运行时，安装程序会优先查找 64 位 JRE，只有当无法找到时，才去查找32位的 JRE。并非强制性规定必须使用安装程序所侦测到的默认 JRE，可以使用任何已经安装的 Java 7 或 更新的 JRE（32 位或 64 位）。
- 托盘图标 当 Tomcat 作为一种服务运行时，不会显示托盘图标。只有当选择在安装完后立即运行 Tomcat 时，不管此时 Tomcat 是否以服务形式运行，托盘图标都会显现。
- 要想更好地了解如何管理以 Windows 服务形式运行的 Tomcat 的信息，可查看 Window 服务指南。

针对启动与配置 Tomcat，安装程序会创建相关的快捷方式。另外，需要特别注意的是，只有当 Tomcat 运行时，Tomcat 的 管理 Web 应用（administration web application）工具才能使用。

UNIX 守护进程

利用 commons-daemon 工程的 jsvc 工具，可以将 Tomcat 作为一个守护进程来运行。Tomcat 的二进制发行版中包含着 jsvc 的源代码包，它需要编译。构建 jsvc 需要一个 C ANSI 编译器（比如 GCC）、GNU Autoconf，以及一个 JDK。

在运行脚本之前，先将环境变量 `JAVA_HOME` 设置为 JDK 的基础路径。在调用 `./configure` 脚本时，需要使用 `--with-java` 参数来指定 JDK 路径，比如：`./configure --with-java=/usr/java`。

使用下列命令应该就能返回一个编译好的 jsvc 二进制可执行文件，位于 `$CATALINA_HOME/bin` 目录中——这需要的前提条件是：使用了 GNU TAR，并且将环境变量 `CATALINA_HOME` 指向 Tomcat 安装基本路径。

请注意，应该使用 GNU make（gmake）而不是 FreeBSD 系统下的原生 BSD make。

```
cd $CATALINA_HOME/bin
tar xvfz commons-daemon-native.tar.gz
cd commons-daemon-1.0.x-native-src/unix
./configure
make
cp jsvc ../..
cd ../..
```

使用下列命令，Tomcat 就可以作为一个守护进程来运行了。

```
CATALINA_BASE=$CATALINA_HOME
cd $CATALINA_HOME
./bin/jsvc \
    -classpath $CATALINA_HOME/bin/bootstrap.jar:$CATALINA_HOME/bin/tomcat-juli.jar
\
    -outfile $CATALINA_BASE/logs/catalina.out \
    -errfile $CATALINA_BASE/logs/catalina.err \
    -Dcatalina.home=$CATALINA_HOME \
    -Dcatalina.base=$CATALINA_BASE \
    -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager \
    -Djava.util.logging.config.file=$CATALINA_BASE/conf/logging.properties \
    org.apache.catalina.startup.Bootstrap
```

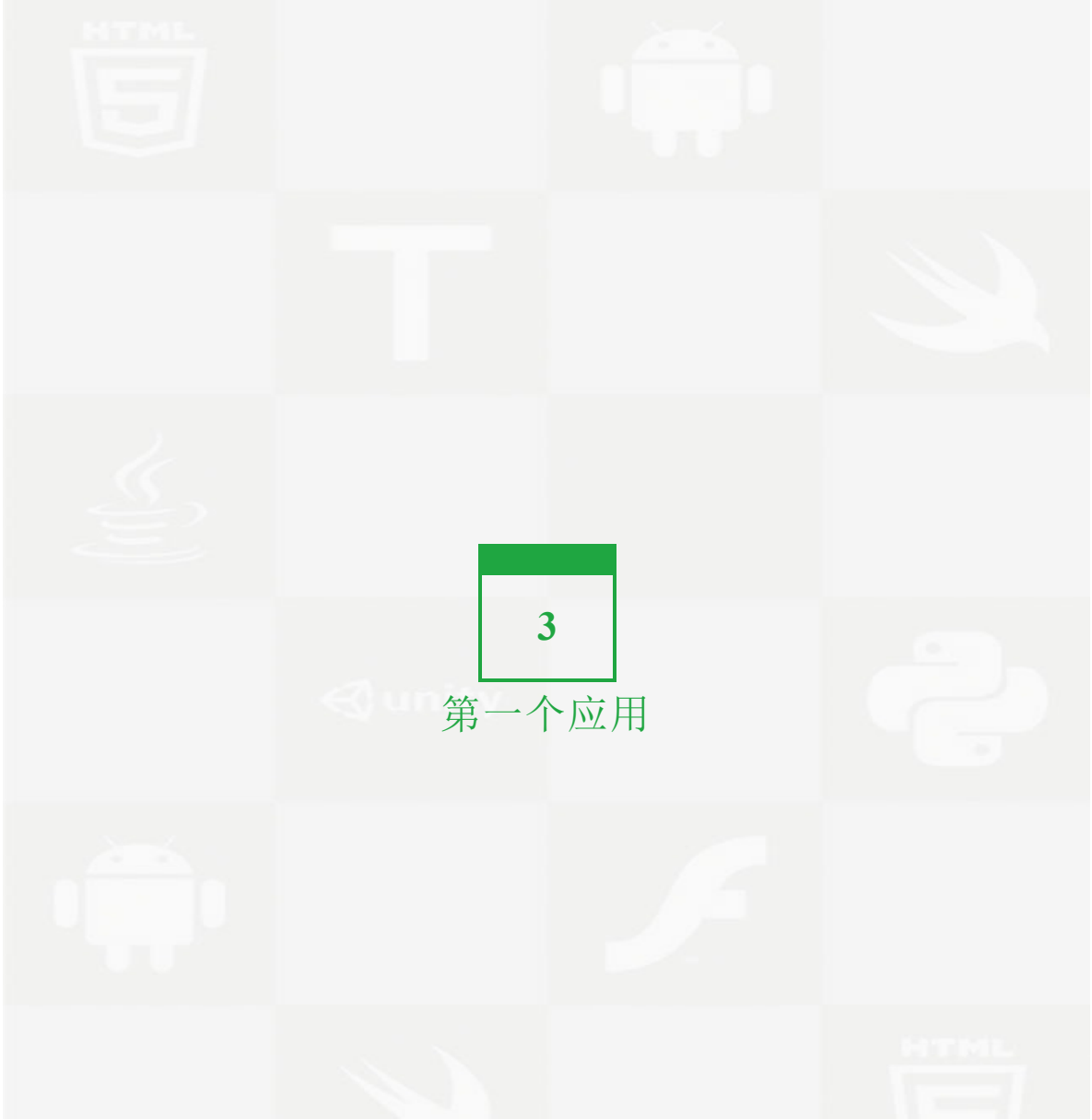
如果 JVM 默认使用的是服务器 VM，而不是客户端 VM，则可能还需要指定 `-jvm server`。这一点已经在 OS X 系统下得到证实。

jsvc 还有其他一些有用的参数。比如：`-user` 就能让守护进程初始化完成后切换到另一个用户，从而能以非特权用户来运行 Tomcat，同时又能使用特权端口。不过要注意的是，如果使用这个选项来以根用户运行 Tomcat，需要禁用 `org.apache.catalina.security.SecurityListener` 检查，这个检查是用来防止以根用户来运行 Tomcat 的。

`jsvc --help` 参数会提供完整的 jsvc 用途信息。尤其是 `-debug` 参数，它对于调试 jsvc 运行中出现的问题是非常有用的一个工具。

`$CATALINA_HOME/bin/daemon.sh` 可以作为一个模板，利用 jsvc `/etc/init.d/` 在启动时自动开启 Tomcat。

注意，要想以上述方式运行 Tomcat，Commons-Daemon JAR 文件必须位于运行时的类路径上。Commons-Daemon JAR 文件在 bootstrap.jar 清单的类路径项中。如果某个 Commons-Daemon 类出现了 ClassNotFoundException（无法找到类） 或 NoClassDefFoundError（无法找到类定义） 这样的错误，那么在加载 jsvc 时将 Commons-Daemon JAR 添加到 `-cp` 参数中。



3 第一个应用

前言

本手册包含了许多 Tomcat 项目开发者论坛成员贡献的内容。下列作者提供了其中的重要内容：

- Craig R. McClanahan（`craigmcc@apache.org`）

目录

本手册由以下各部分构成：

4

Tomcat Web 应用部署

本章概述

部署这个术语描述的就是，将 Web 应用（第三方的 WAR 文件，或是你自己定制的 Web 应用）安装到 Tomcat 服务器上的整个过程。

在 Tomcat 服务器上，可以通过多种方法部署 Web 应用：

- 静态部署。在启动 Tomcat 之前安装 Web 应用。
- 动态部署。使用 Tomcat 的 Manager 应用直接操控已经部署好的 Web 应用（依赖 auto-deployment 特性）。

Tomcat Manager 是一种能交互使用（利用 HTML GUI）的 Web 应用，还可以利用编程的方式（通过基于 URL 的 API）来部署并管理 Web 应用。

依靠 Manager 这种 Web 应用，可以实施多种部署。Tomcat 为 Apache Ant 构建工具提供了多个任务。 Apache Tomcat Maven Plugin 工程则提供了与 Apache Maven 的集成。另外还有一种工具叫做客户端配置器 （Client Deployer，TCD），它通过命令行来使用，提供一些额外的功能，比如编译与验证 Web 应用，以及将 Web 应用打包成 Web 应用资源（WAR）文件。

安装

静态部署 Web 应用时，并不需要附加的安装，因为 Tomcat 已经提供了这项功能。利用 Tomcat Manager 部署应用也不需要任何安装，不过需要进行一番配置，详见Tomcat Manager 手册。如果使用客户端配置器的话，就必须要进行安装了。

Tomcat 的核心分发版并不包括 TCD，必须从下载区独立下载它，下载文件通常冠名为：*apache-tomcat-8.0.x-deployer*。

要想使用 TCD，必须事先配置有 Apache Ant 1.6.2+ 以及 Java 安装。另外，还必须定义一个指向 ANT 安装根目录的 ANT_HOME 环境变量，以及一个指向 Java 安装目录的 JAVA_HOME 值。另外，还必须确保必须在操作系统所提供的命令 shell 中运行 ANT 的 `ant` 命令，以及 Java 的 `javac` 编译器命令。

1. 下载 TCD 分发版；
2. TCD 包可以解压缩到任何地方，不需要解压缩到任何已存在的 Tomcat 安装下。
3. 相关用法，参考下文的使用客户端部署器进行部署

关于上下文

在谈到 Web 应用的配置时，需要理解一下上下文（Context）这个概念。上下文在 Tomcat 中其实就是 Web 应用的意思。

为了在 Tomcat 中配置上下文，需要用到上下文描述符文件（ Context Descriptor）。上下文描述符文件其实就是一个 XML 文件，含有 Tomcat 与上下文相关的配置信息，例如命名资源或会话管理器配置信息。在 Tomcat 的早期版本中，上下文描述符文件配置的内容经常保存在 Tomcat 的主要配置文件 server.xml 中，但现在不再推荐采用这一方式（虽然目前它依然有效）。

上下文描述符文件不仅能帮助 Tomcat 了解如何配置上下文，而且其他工具（如 Manager 与 TCD）也经常会借助上下文描述符文件来正确履行它们的职责。

上下文描述符文件位于：

1. `$CATALINA_BASE/conf/[enginename]/[hostname]/[webappname].xml`
2. `$CATALINA_BASE/webapps/[webappname]/META-INF/context.xml`

在目录 1 中的文件名为 `[webappname].xml`，但在目录 2 中，文件名为 context.xml。如果某个 Web 应用没有相应的上下文描述符文件，Tomcat 就会使用默认值配置该应用。

在 **Tomcat** 启动时进行部署

如果你对使用 Manager 或 TCD 不是很感兴趣，那就需要先把 Web 应用静态地部署到 Tomcat 中，然后再启动 Tomcat。 这种情况下应用部署的位置由 `appBase` 目录属性来决定，每台主机都指定有这样一个位置。该位置既可以放入未经压缩的 Web 应用资源文件（通常被称为 exploded web application，“膨胀 Web 应用”），也可以放置已压缩过的 Web 应用资源文件（.WAR 文件）。

再次解释一下，应用部署位置由主机（默认主机为 localhost）的 `appBase` 属性来指定。默认的 `appBase` 属性所指定的目录为 `$CATALINA_BASE/webapps`。只有当主机的 `deployOnStartup` 属性为 `true`, 应用才会在 Tomcat 启动时进行自动部署。

在以上情况下，当 Tomcat 启动时，部署的具体步骤如下：

1. 先部署上下文描述符文件。
2. 然后再对没被任何上下文描述符文件引用过的膨胀 Web 应用进行部署。 如果在 `appBase` 中已存在与这种应用有关的 .WAR 文件，而且要比膨胀应用文件更新，那么就会将膨胀应用的文件夹清除，转而从 .WAR 文件中部署 Web 应用。
3. 部署 .WAR 文件。

在运行中的 **Tomcat** 服务器上进行动态应用部署

除了静态部署之外，也可以在运行中的 Tomcat 服务器上进行应用部署。

如果主机的 `autoDeploy` 属性为 `true`，主机就会在必要时尝试着动态部署并更新 Web 应用。 例如，当把一个新 .WAR 文件放入 `appBase` 所指定的名录时。为了实现这种操作，主机就需要启用后台处理，当然这也是默认的配置。

当 `autoDeploy` 设置为 `true` 时，运行中的 Tomcat 服务器能够允许实现以下行为：

- 对放入主机 `appBase` 指定目录下的 .WAR 文件进行配置。
- 对放入主机的膨胀 Web 应用进行配置。
- 对于已通过 .WAR 文件配置好的应用，如果又提供了更新的 .WAR 文件，则使用新 .WAR 文件对该应用重新进行配置。在这种情况下，会先移除原有的膨胀 Web 应用，然后再次对 .WAR 文件进行扩展（膨胀）。注意，如果在主机配置中，没有把 `unpackWARs` 属性设为 `false`，则 WAR 文件将不会膨胀，这时 Web 应用将部署为一个压缩文档。
- 如果 `/WEB-INF/web.xml` 文件（或者任何其他被定义为 WatchedResource 的资源）更新，则重新加载 Web 应用。
- 如果用来部署 Web 应用的上下文描述符更新，则重新部署该 Web 应用。
- 如果 Web 应用所使用的全局或者每台主机中的上下文描述符已更新，则重新部署与该应用有依赖关系的 Web 应用。
- 如果一个上下文描述符被添加到 `$CATALINA_BASE/conf/[enginename]/[hostname]/` 目录中，并且该描述文件带有与之前部署的 Web 应用的上下文路径相对应的文件名，则重新部署该 Web 应用。
- 如果某个 Web 应用的文档库（docBase）被删除，则取消对该应用的部署。注意，在 Windows 系统下，要想实现这样的行为，必须开启防锁死功能（参看 Context 配置文档），否则无法删除运行中的 Web 应用的资源文件。

> 注意，也可以在加载器中对 Web 应用的重新加载进行配置，在这种情况下，会跟踪已加载的类所产生的更改。

使用 **Tomcat Manager** 进行部署

详情参看 Tomcat Manager 文档。

使用客户端部署器进行部署

最后要介绍的是利用客户端部署器（TCD）对 Web 应用进行部署。客户端部署器可以实施的行为包括：验证并编译 Web 应用，将资源文件压缩成 .WAR 文件，并将 Web 应用部署到用于生产或开发环境的 Tomcat 服务器上。一定要注意，该特性的实现需要使用 Tomcat Manager，而且目标 Tomcat 服务器也应处于运行状态。

因为会用到 TCD，所以要求用户还必须熟悉 Apache Ant。Apache Ant 是一个脚本编译工具。TCD 每个包都会带有一个编译脚本。只需大体能够了解 Apache Ant 即可（本节前面列有其安装细则，这里需要熟练使用操作系统命令 shell 以及配置环境变量）。

TCD 包括一些 Ant 任务，在配置前用于 JSP 编译的 Jasper 页面编译器，以及验证 Web 应用上下文描述符的任务。验证器任务（`org.apache.catalina.ant.ValidatorTask`类）只允许传入一个参数：膨胀 Web 应用的基本路径。

TCD 使用膨胀 Web 应用作为输入（下面列出了其所用的属性列表）。通过部署器，以编程方式部署的 Web 应用可能会在 /META-INF/context.xml 中包含一个上下文描述符。

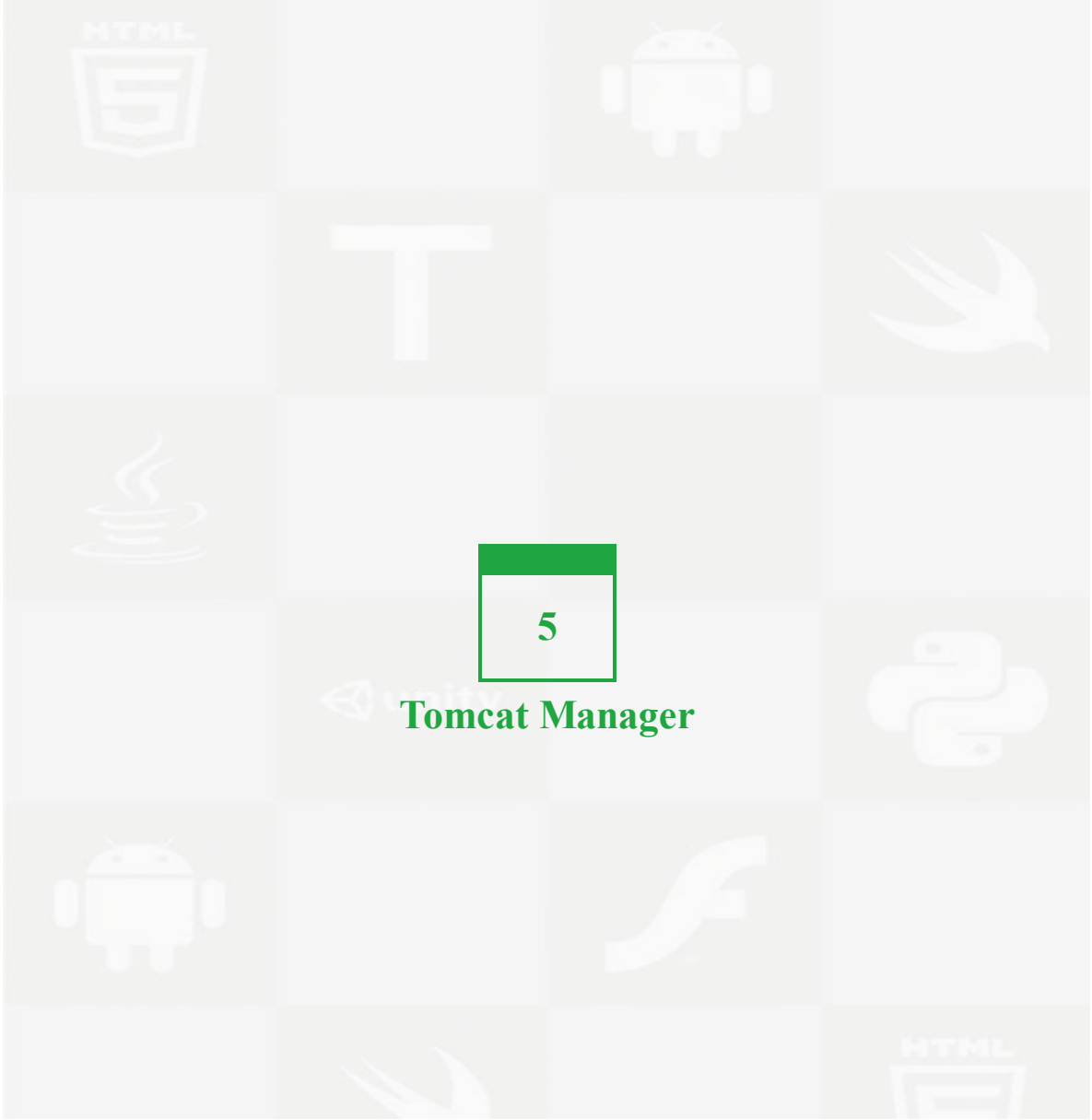
TCD 包含一个可即时使用的 Ant 脚本， 其中包含以下目标。

- `compile`（默认）：编译并验证 Web 应用。可以单独使用，并不需要运行着的 Tomcat 服务器。已编译的应用只能运行在相关的 Tomcat X.Y.Z 版本的服务器上，又因为 Jasper 生成的代码依赖它的运行时组件，所以已编译应用并不一定能在其他版本的 Tomcat 版本上运行。另外值得注意的是，该目标也能自动编译位于`/WEB-INF/classes`这一应用目录下的任何 Java 源文件。
- `deploy` 在 Tomcat 服务器上部署 Web 应用（无论其是否编译过）。
- `undeploy` 取消对某个 Web 应用的部署。
- `start` 开启 Web 应用。
- `reload` 重新加载 Web 应用。
- `stop` 停止 Web 应用。

为了能够配置部署，还需要在 TCD 安装的根目录下创建一个叫做`deployer.properties`的文件，并在该文件中的每行添加下列名值对：

除此之外，你还必须确定为 TCD 所使用的目标 Tomcat Manager 创建了一个用户，否则 TCD 就无法验证 Tomcat Manager，从而造成配置失败，详细信息参看 Tomcat Manager 文档。

- **`build`** build 文件夹默认位置是`${build}/webapp/${path}`（其中`${build}`的默认指向位置是`${basedir}/build`）。`compile`目标执行完毕后，Web 应用的 .WAR 文件将位于`${build}/webapp/${path}.war`。
- **`webapp`** 该文件夹包含后续将进行编译与验证的膨胀 Web 应用。默认情况下，该文件夹是`myapp`。
- **`path`** Web 应用已部署的上下文路径，默认为`/myapp`。
- **`url`** 指向运行中的 Tomcat 服务器中的某个 Tomcat Manager Web 应用的绝对路径，用于对 Web 应用的部署与取消部署。默认情况下，部署器会尝试访问运行在 localhost 上的 Tomcat 实例，其 url 为`http://localhost:8080/manager/text`。
- **`username`** Tomcat Manager 的用户名（用户应具有读写 manager-script 的权限）。
- **`password`** Tomcat Manager 的密码。



5

Tomcat Manager

概述

很多生产环境都非常需要以下特性：在无需关闭或重启整个容器的情况下，部署新的 Web 应用或者取消对现有应用的部署。或者，即便在 Tomcat 服务器配置文件中没有指定 `reloadable` 的情况下，也可以请求重新加载现有应用。

Tomcat 中的 Web 应用 Manager 就是来解决这些问题的，它默认安装在上下文路径：`/manager` 中，支持以下功能：

- 用已上传的 WAR 文件内容部署新的 Web 应用。
- 在服务器文件系统中指定上下文路径处部署新的 Web 应用。
- 列出当前已部署的 Web 应用，以及这些应用目前的活跃会话。
- 重新加载现有的 Web 应用，以便响应 `/WEB-INF/classes` 或 `/WEB-INF/lib` 中内容的更改。
- 列出操作系统及 JVM 的属性值。
- 列出可用的全局 JNDI 资源，它们将用于预备 `<ResourceLink>` 元素的部署工具中。`<ResourceLink>` 元素内嵌于 `<Context>` 部署描述中。
- 开启一个已停止的 Web 应用，从而使其再次可用。
- 停止一个现有的 Web 应用，从而使其不可用，但并不取消对它的部署。
- 取消对一个已部署 Web 应用的部署，删除它的文档库目录（除非它是从文件系统中部署的）。

Tomcat 默认安装已经包含了 Manager。 将一个 Manager 应用实例的 `Context` 添加到一个新的主机中，`manager.xml` 上下文配置文件应放在 `$CATALINA_BASE/conf/[enginename]/[hostname]` 文件夹中。如下所示：

```
<Context privileged="true" antiResourceLocking="false"
         docBase="${catalina.home}/webapps/manager">
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
         allow="127\.0\.0\.1" />
</Context>
```

如果将 Tomcat 配置成能够支持多个虚拟主机（网站），则需要对每个虚拟主机配置一个 Manager。

Manager 应用的使用方式有以下三种：

- 作为带有用户界面的应用，在浏览器中运行。在随后这个范例 URL 中，你可以将 `localhost` 替换为你的网站主机名称：`http://localhost:8080/manager/html`。
- 只使用 HTTP 请求的一个功能最少的版本。它适合系统管理员通过创建脚本来进行使用。将命令指定在请求的 URI 中，响应是简单格式的文本（易于解析与处理）。详情查看 支持的 Manager 命令。
- 用于 Ant 构建工具（1.4或更新版本）的一套方便的任务定义。详情参见 利用 Ant 执行 Manager 命令。

配置 Manager 应用访问

> 下文的描述将使用变量名 `$CATALINA_BASE` 来指代工作目录。如果你还没有为多个 Tomcat 实例设置 CATALINA_BASE 目录，那么 `$CATALINA_BASE` 就将被设置为 `$CATALINA_HOME`（Tomcat 的安装目录）的值。

Tomcat 以默认值运行是非常危险的，因为这能让互联网上的任何人都可以在你的服务器上执行 Manager 应用。因此，Manager 应用要求任何用户在使用前必须验证自己的身份，提供自己的用户名和密码，以及相应配置的 **manager-**** 角色（角色名称根据所需的功能而定）。另外，默认用户文件（`$CATALINA_BASE/conf/tomcat-users.xml`）中的用户名称都没有指定角色名称，所以默认是不能访问 Manager 应用的。

这些角色名称位于 Manager 应用的 `web.xml` 文件中。可用的角色包括：

- **manager-gui** 能够访问 HTML 界面。
- **manager-status** 只能访问“服务器状态”（Server Status）页面。
- **manager-script** 能够访问文档中描述的适用于工具的纯文本界面，以及“服务器状态”页面。
- **manager-jmx** 能够访问 JMX 代理界面以及“服务器状态”（Server Status）页面。

HTML 界面不会遭受 CSRF（Cross-Site Request Forgery，跨站请求伪造）攻击，但纯文本界面及 JMX 界面却有可能无法幸免。这意味着，如果用户能够访问纯文本界面及 JMX 界面，那么在利用 Web 浏览器去访问 Manager 应用时，必须要万分谨慎。要想保持对 CSRF 免疫，则必须：

- 在使用 Web 浏览器访问 Manager 应用时，假如用户具有 **manager-script** 或 **manager-jmx** 角色（比如为了测试纯文本界面或 JMX 界面），那么必须关闭所有的浏览器窗口，终止会话。如果不关闭浏览器，访问了其他站点，就可能会遭受 CSRF 攻击。
- 建议永远不要将 **manager-script** 或 **manager-jmx** 角色授予那些拥有 **manager-gui** 角色的用户。

> 注意 JMX 代理界面是 Tomcat 中非常高效的底层、类似于根级别的管理界面。如果用户知道了该调用的命令，就能实现大量行为，所以一定不要轻易授予用户 **manager-jmx** 角色。

为了能够访问 Manager 应用，必须创建一个新的用户名/密码组合，并为之授予一种 **manager-**** 角色，或者把一种 **manager-**** 角色授予现有的用户名/密码组合。因为本文档的大部分内容都在描述纯文本界面的命令，所以为了将来讨论实例方便起见，把角色名称定为 **manager-script**。而涉及到具体如何配置用户名及密码，则是跟你所使用的Realm 实现有关：

- *UserDatabaseRealm* 加上 *MemoryUserDatabase* 或 *MemoryRealm*——*UserDatabaseRealm* 和 *MemoryUserDatabase* 配置在默认的 `$CATALINA_BASE/conf/server.xml` 文件中。*MemoryUserDatabase* 和 *MemoryRealm* 都会读取储存在 `CATALINA_BASE/conf/tomcat-users.xml` 里的 XML 格式文件，它可以用任何文本编辑器进行编辑。该文件会为每个用户定义一个 XML 格式的 `<user>`，如下所示：

  ```
  <user username="craigmcc" password="secret" roles="standard,manager-script" />
  ```

  它定义了用户登录时所用的用户名和密码，以及他或她采用的角色名称。你可以把 **manager-script** 角色添加到由逗号分隔的 `roles` 属性中，从而将该角色赋予一个或多个用户，也可以利用指定角色来创建新的用户。

- *DataSourceRealm* 或 *JDBCRealm*——用户和角色信息都存储在一个经由 JDBC 访问的数据库中。按照当前环境的标准流程，将 **manager-script** 角色赋予一个或多个用户，或者利用该角色创建一个或多个新用户。

- *JNDIRealm*——你的用户和角色信息被存储在经由 LDAP 访问的一个目录服务器中。按照当前环境的标准流程，为一个或更多的现有用户添加 **manager-script** 角色，和（/或）利用指定角色创建一个或更多的新用户。

在下一节，当你第一次尝试使用 Manager 的一个命令时，将会使用基本验证进行登录。用户名和密码的具体内容并不重要，只要它们能够证明，用户数据库中拥有 **manager-script** 角色的用户是有效用户，我们的目的就达到

了。

除了密码限制访问之外，Manager 还可以配置 `RemoteAddrValve` 和 `RemoteHostValve` 这两个参数，分别通过远程 IP 地址或远程主机名来进行限制访问。详情可查看 Valve 文档。下列范例是通过 IP 地址来限制访问本地主机：

```
<Context privileged="true">
         <Valve className="org.apache.catalina.valves.RemoteAddrValve"
                allow="127\.0\.0\.1"/>
</Context>
```

易用的 **HTML** 界面

Manager 应用易用的 HTML 界面位于：

```
http://{host}:{port}/manager/html
```

正像前面讲过的那样，需要被授予 **manager-gui** 角色才能访问它。关于这个界面，还有一个独立的文档，请访问以下页面：

- Manager 的 HTML 界面文档

HTML 界面可免受 CSRF（跨站请求伪造）攻击。对 HTML 页面的每次访问都会生成一个随机令牌，存储在会话中，包含在页面的所有链接中。如果你的下一个操作没有正确的令牌值，操作就会被拒绝。如果令牌过期，可以从主页或者 Manager 的 *List Applications*（列出的应用）页面重新开始。

Manager 支持的命令

Manager 应用能够处理的命令都是通过下面这样的请求 URL 来指定的：

```
http://{host}:{port}/manager/text/{command}?{parameters}
```

`{host}` 和 `{port}` 分别代表运行 Tomcat 服务器所在的主机名和端口号。`{command}` 代表所要执行的 Manager 命令。`{parameters}` 代表该命令所专有的查询参数。在后面的实例中，可以为你的安装自定义适当的主机和端口。

这些命令通常是被 HTTP `GET` 请求执行的。`/deploy` 命令有一种能够被 HTTP `PUT` 请求所执行的形式。

常见参数

多数 Manager 命令都能够接受一个或多个查询参数，这些查询参数如下所示：

- **path** 要处理的 Web 应用的上下文路径（包含前面的斜杠）。要想选择 ROOT Web 应用，指定 `/` 即可。注意：无法对 Manager 应用自身执行管理命令。
- **version** 并行部署 所用的 Web 应用版本号。
- **war** Web 应用归档（WAR）文件的 URL，或者含有 Web 应用的目录路径名，或者是上下文配置 .xml 文件。你可以按照以下任一格式使用 URL：
  - **file:/absolute/path/to/a/directory** 解压缩后的 Web 应用所在的目录的绝对路径。它将不做任何改动，直接附加到你所指定的上下文路径上。
  - **file:/absolute/path/to/a/webapp.war** Web 应用归档（WAR）文件的绝对路径。只对 `/deploy` 命令有效，也是该命令所唯一能接受的格式。
  - **jar:file:/absolute/path/to/a/warfile.war!/** 本地 WAR 文件的 URL。另外，为了能够完整引用一个JAR 文件，你可以采用 `JarURLConnection` 类的任何有效语法。
  - **file:/absolute/path/to/a/context.xml** Web 应用上下文配置 .xml 文件的绝对路径，上下文配置文件包含着上下文配置元素。
  - **directory** 主机应用的基本目录中的 Web 应用的目录名称。
  - **webapp.war** 主机应用的基本目录中的 WAR 文件的名称。

每个命令都会以 `text/plain` 的形式（比如，没有 HTML 标记的纯 ASCII 码文本 ）返回响应，从而便于开发者与程序阅读。响应的第一行以 `OK` 或 `FAIL` 开头，表明请求的命令是否成功。如果失败，响应第一行随后部分就会带有遇到问题的描述。一些包含其他信息行的命令会在下文予以介绍。

国际化说明 Manager 应用会在资源包中查找消息字符串，所以这些字符串可能已经转化为你所用平台的语言版本了。下文的范例展示的全都是消息的英文版本。

远程部署新应用

`http://localhost:8080/manager/text/deploy?path=/foo`

将作为请求数据指定在 HTTP `PUT` 请求中的 Web 应用归档文件（WAR）上传，将它安装到相应虚拟主机的 `appBase` 目录中，启动，使用目录名或不带 .war 后缀的 WAR 文件名作为路径。稍后，可以通过 `/undeploy` 取消对应用的部署，相应的应用目录也会被删除。

该命令通过 HTTP `PUT` 请求来执行。

通过在 `/META-INF/context.xml` 中包含上下文配置 XML 文件，.WAR 文件能够包含 Tomcat 特有的部署配置信息。

URL 参数包括：

- `update` 设置为 true 时，任何已有的更新将会首先取消部署。默认值为 false。
- `tag` 指定一个标签名称。这个参数能将已部署的 Web 应用与标签连接起来。如果 Web 应用被取消部署，则以后在需要重新部署时，只需使用标签就能实现。

注意：该命令是 `/undeploy` 命令在逻辑上是对立的。

如果安装或启动成功，会接受到这样一个响应：

```
OK - Deployed application at context path /foo
```

否则，响应会以 `FAIL` 开始，并包含一个错误消息。出现问题的可能原因为：

- *Application already exists at path /foo*
  当前运行的 Web 应用的上下文路径必须是唯一的。否则，必须使用这一上下文路径取消对现有 Web 应用的部署，或者为新应用选择另外一个上下文路径。`update` 参数可以指定为 URL 中的参数。true 值可避免这种错误。这种情况下，会在部署前，取消对现有应用的部署。

- *Encountered exception*
  遇到试图开启新的 Web 应用。可查看 Tomcat 日志了解详情。但有可能是在解析 `/WEB-INF/web.xml` 文件时遇到了问题，或者在初始化应用的事件侦听器与过滤器时出现遗失类的情况。

从本地路径处部署新的应用

部署并启动一个新的 Web 应用，附加到指定的上下文 `path` 上（不能被其他 Web 应用同时使用）。该命令与 `/undeploy` 在逻辑上是对立的。

该命令由一个 HTTP `GET` 请求执行。部署命令的应用方式有很多种。

部署之前部署过的 **Web** 应用

```
http://localhost:8080/manager/text/deploy?path=/footoo&tag=footag
```

用来部署之前曾通过 `tag` 属性部署过的 Web 应用。注意，Manager 应用的工作目录包含之前部署过的 WAR 文件；如果清除它则将使部署失败。

通过 **URL** 部署一个目录或 **WAR** 文件

部署位于 Tomcat 服务器上的 Web 应用目录或 .war 文件。如果没有指定上下文路径参数 `path`，就会把目录名或未带 .war 后缀的 war 文件名当做路径来使用。`war` 参数指定了目录或 WAR 文件的 URL（也包含 `file:` 格式）。引用 WAR 文件的 URL 所采用的语法详见 `java.net.JarURLConnection` 类的 Java 文档页面。只使用引用了整个 WAR 文件的 URL。

下面这个实例中，Web 应用位于 Tomcat 服务器上的 `/path/to/foo` 目录中，被部署为上下文路径为 `/footoo` 的 Web 应用。

```
http://localhost:8080/manager/text/deploy?path=/footoo&war=file:/path/to/foo
```

在下例中，Tomcat 服务器上的 .war 文件 `/path/to/bar.war` 被部署为上下文路径为 `/bar` 的 Web 应用。注意，这里没有 `path` 参数，因此上下文路径默认为没有 .war 后缀的 WAR 文件名。

```
http://localhost:8080/manager/text/deploy?war=jar:file:/path/to/bar.war!/
```

从主机的 **appBase** 目录中部署一个目录或 **WAR**

对位于主机 `appBase` 目录中的 Web 应用目录或 .war 文件进行部署。目录名或没有 .war 后缀名的 WAR 文件名被用作上下文路径名。

在下面的范例中，Web 应用位于 Tomcat 服务器中主机 `appBase` 目录下名为 `foo` 的子目录中，被部署为上下文路径名为 `/foo` 的 Web 应用。注意，用到的上下文路径名就是 Web 应用的目录名。

```
http://localhost:8080/manager/text/deploy?war=foo
```

在下面的范例中，位于主机 `appBase` 目录中的 `bar.war` 文件被部署为上下文名为 `/bar` 的 Web 应用。

```
http://localhost:8080/manager/text/deploy?war=bar.war
```

使用上下文配置 **.xml** 文件来进行部署

如果主机的 `deployXML` 标志设定为 true，就可以使用上下文配置 .xml 文件以及一个可选的 .war 文件（或 Web 应用目录）来进行 Web 应用部署。在使用上下文 .xml 文件配置文件进行部署时，不会用到上下文路径参数 `/path`。

上下文配置 .xml 文件包含用于 Web 应用上下文的有效 XML，就好像是在 Tomcat 的 `server.xml` 配置文件中进行配置一样。范例如下：

```
<Context path="/foobar" docBase="/path/to/application/foobar">
</Context>
```

可选的 `war` 参数被设定为指向 Web 应用的 .war 文件或目录的 URL，它会覆盖掉上下文配置 .xml 文件中的任意 `docBase`。

在下面这个实例中，使用上下文配置 .xml 文件部署 Web 应用：

```
http://localhost:8080/manager/text/deploy?config=file:/path/context.xml
```

在下面这个应用部署范例中，使用了上下文配置 .xml 文件和位于服务器中的 Web 应用的 .war 文件。

```
http://localhost:8080/manager/text/deploy
  ?config=file:/path/context.xml&war=jar:file:/path/bar.war!/
```

部署中的一些注意事项

如果主机配置中将 `unpackWARs` 设为 true，而且你部署了一个 war 文件，那么这个 war 文件将解压缩至主机的 `appBase` 目录下的一个目录中。

如果应用的 war 文件或目录安装在主机的 `appBase` 目录中，那么或者主机应该被部署为 `autoDeploy` 为 true，或者上下文路径必须匹配目录名或不带 .war 后缀的 war 文件名。

为了避免不可信用户作出对 Web 应用的侵害，主机的 `deployXML` 标志可以设为 false。这能保证不可信用户通过使用 XML 配置文件来部署 Web 应用，也能阻止他们部署位于其主机 `appBase` 之外的应用目录或 .war 文件。

部署响应

如果安装及启动都正常，会得到以下这样的响应：

```
OK - Deployed application at context path /foo
```

否则，响应会以 `FAIL` 开头并包含一些错误消息，引起问题的原因可能有以下几种：

- *Application already exists at path /foo*
  当前运行的 Web 应用的上下文路径必须是唯一的。否则，必须使用这一上下文路径取消对现有 Web 应用的部署，或者为新应用选择另外一个上下文路径。`update` 参数可以指定为 URL 中的参数。true 值可避免这种错误。这种情况下，会在部署前，取消对现有应用的部署。

- *Document base does not exist or is not a readable directory*
  通过 `war` 指定的 URL 必须要确认服务器中的某个目录含有解压缩后的 Web 应用，包含该应用的 WAR 文件的绝对 URL。更正 `war` 参数所提供的值。

- *Encountered exception*
  遇到试图开启新 Web 应用。可查看 Tomcat 日志了解详情。但有可能是在解析 `/WEB-INF/web.xml` 文件时遇到了问题，或者在初始化应用的事件侦听器与过滤器时出现遗失类的情况。

- *Invalid application URL was specified* 所指定的指向目录或 Web 应用的 URL 无效。有效的 URL 必须以 `file:` 开始，用于 WAR 文件的 URL 必须以 .war 结尾。

- *Invalid context path was specified*
  上下文路径必须以斜杠字符开始，引用 ROOT 应用必须使用 `/`。

- *Context path must match the directory or WAR file name*

如果应用的 .war 文件或目录安装在主机的 `appBase` 目录，那么或者主机应该被部署为 `autoDeploy` 为 true，或者上下文路径必须匹配目录名或不带 .war 后缀的 war 文件名。

- *Only web applications in the Host web application directory can be installed* 如果主机的 `deployXML` 标志为设为 false，那么当要部署的 Web 应用目录或 .war 文件位于主机 `appBase` 目录之外时，就会产生这样的错误。

列出当前已部署的应用

`http://localhost:8080/manager/text/list`

列出当前所有部署的 Web 应用的上下文路径、当前状态（`running` 或 `stopped`），以及活跃会话。开启 Tomcat 后，一般立刻会产生如下这样的响应：

```
OK - Listed applications for virtual host localhost
/webdav:running:0
/examples:running:0
/manager:running:0
/:running:0
```

Listed applications for virtual host localhost：列出虚拟主机本地主机的所有应用。

重新加载一个现有应用

```
http://localhost:8080/manager/text/reload?path=/examples
```

标记一个现有应用，关闭它并重新加载。这一功能的适用情况为：当 Web 应用上下文不能重新加载；你已经更新了 `/WEB-INF/classes` 目录中的类和属性文件时；或者当你在 `/WEB-INF/lib` 目录添加或更新了 jar 文件。

> 注意：在重新加载时，Web 应用配置文件 `/WEB-INF/web.xml` 无法重新读取。如果对 web.xml 文件作出改动，则必须停止并启动 Web 应用。

如果命令成功执行，应得如下所示的响应：

```
OK - Reloaded application at context path /examples
```

否则，返回的响应以 `FAIL` 开头，并包含相关的错误消息。引起问题的可能原因有以下几种：

- *Encountered exception*
  遇到试图重启 Web 应用的异常。可查看 Tomcat 日志了解详情。
- *Invalid context path was specified*
  上下文路径必须以斜杠开始，引用 ROOT Web 应用必须使用 `/`。
- *No context exists for path /foo*
  在所指定的上下文路径中没有发现部署好的应用。
- *No context path was specified*
  需要 `path` 参数。
- *Reload not supported on WAR deployed at path /foo*
  当前，如果主机配置为不解压缩 WAR 文件时，直接从一个 WAR 文件安装 Web 应用时，不支持重新加载应用（以便使类或 web.xml 文件中的更改生效）。由于只有在从已解压缩目录安装 Web 应用时才生效，所以在使用 WAR 文件时，应该先取消对应用的部署，然后重新部署该应用，以便使更改生效。

列出 **OS** 及 **JVM** 属性

`http://localhost:8080/manager/text/serverinfo`

列出 Tomcat 版本、操作系统以及 JVM 属性等相关信息。

如果出现错误，响应会以 `FAIL` 开始并包含一系列错误消息，导致错误的可能原因包括有：

- *Encountered exception*
  碰到异常试图列举系统属性。可查看 Tomcat 日志了解详情。

列出可能的全局 **JNDI** 资源

`http://localhost:8080/manager/text/resources[?type=xxxxx]`

列出上下文配置文件资源链接中所使用的全局 JNDI 资源。如果指定 `type` 请求参数，参数值必须是所需资源类型的完整 Java 类名（比如，指定 `javax.sql.DataSource` 获取所有可用的 JDBC 数据资源的名称）。如果没有指定 `type` 请求参数，则将返回所有类型的资源。

根据是否指定了 `type` 请求参数，常见响应的第一行将如下所示：

```
OK - Listed global resources of all types
```

或

```
OK - Listed global resources of type xxxxx
```

后面将每个资源都单列一行，每一行内的字段都由冒号（`:`）分隔，如下所示：

- *Global Resource Name* 全局 JNDI 资源的名称，将用在 `<ResourceLink>` 元素的 `global` 属性中。
- *Global Resource Type* 该全局 JNDI 资源的完整描述的 Java 类名。

如果出现错误，响应将会以 `FAIL` 开始，并包含一个错误消息。出错的原因可能包括以下几方面：

- *Encountered exception*
  碰到异常试图列举 JNDI 资源，可查看 Tomcat 日志了解详情。
- *No global JNDI resources are available*
  运行的 Tomcat 服务器没有配置全局 JNDI 资源。

会话统计

`http://localhost:8080/manager/text/sessions?path=/examples`

显示 Web 应用默认的会话超时，当前活跃会话在一分钟范围内实际的超时次数。比如，重启 Tomcat 并随后执行 `/examples` Web 应用中的一个 JSP 范例，有可能得到如下信息：

```
OK - Session information for application at context path /examples
Default maximum session inactive interval 30 minutes
<1 minutes: 1 sessions
1 - <2 minutes: 1 sessions
```

过期会话

```
http://localhost:8080/manager/text/expire?path=/examples&idle=num
```

显示会话统计信息（比如上面的 `/sessions` 命令）以及超出 `num` 所指定的分钟数的过期会话。要想使所有会话都过期，可使用 `&idle = 0`。

```
OK - Session information for application at context path /examples
Default maximum session inactive interval 30 minutes
1 - <2 minutes: 1 sessions
3 - <4 minutes: 1 sessions
>0 minutes: 2 sessions were expired
```

实际上，`/sessions` 和 `/expire` 是同一个命令的两种异名，唯一不同之处在于 `idle` 参数。

开启一个现有应用

```
http://localhost:8080/manager/text/start?path=/examples
```

标记一个已停止的应用，重新开启它，使其再次可用。停止并随后重新开启应用有时显得非常重要，比如当应用所需的服务器暂时变得不可用时。通常情况下，与其让用户频繁碰到数据库异常，倒不如停止基于该数据库的 Web 应用运行。

如果该命令成功执行，将得到类似如下的响应：

```
OK - Started application at context path /examples
```

否则，将返回出错响应，该响应以 `FAIL` 开头，并包含一些错误。出错原因可能是由于：

- *Encountered exception*
  碰到异常情况，试图开启 Web 应用。可检查 Tomcat 日志了解详情。
- *Invalid context path was specified*
  上下文路径必须以斜杠字符开始，引用 ROOT Web 应用必须使用反斜杠（`/`）。
- *No context exists for path /foo*
  在指定的上下文路径处没有部署的应用。
- *No context path was specified*
  需要指定 `path` 参数。

停止已有应用

```
http://localhost:8080/manager/text/stop?path=/examples
```

标记现有应用，使其不可用，但仍使其处于已部署状态。当应用停止时，任何请求都将得到著名的 HTTP 404错误。在应用列表中，该应用将显示为“stopped”。

如果该命令成功执行，将得到类似如下的响应：

```
OK - Stopped application at context path /examples
```

否则，将返回出错响应，它以 `FAIL` 开头，并包含一个出错消息，可能导致出误的原因包括：

- *Encountered exception*
  碰到异常情况，试图开启 Web 应用。可检查 Tomcat 日志了解详情。
- *Invalid context path was specified*
  上下文路径必须以斜杠字符开始，引用 ROOT Web 应用必须使用反斜杠（`/`）。
- *No context exists for path /foo* 在指定的上下文路径处没有部署的应用。
- *No context path was specified*
  需要指定 `path` 参数。

取消对现有应用的部署

```
http://localhost:8080/manager/text/undeploy?path=/examples
```

警告：该命令将删除虚拟主机 **appBase** 目录（通常是 **webapps** ）中的所有 **Web** 应用。该命令将从未解压缩（或已解压缩）的 .WAR 式部署中，以及 `$CATALINA_BASE/conf/[enginename]/[hostname]/` 中以 XML 格式保存的上下文描述符中，删除应用的 .WAR 文件及目录。如果你只是想让某个应用暂停服务，则应该使用 `/stop` 命令。

标记一个已有的应用，将其恰当地关闭，从 Tomcat 中移除（从而使得以后可以重新使用该上下文路径）。另外，如果文档根目录位于虚拟主机的 `appBase` 目录（通常是 webapps）中，则它也将被移除。该命令是 `/deploy` 的逆向命令。

如果该命令成功执行，将得到类似如下的响应：

```
OK - Undeployed application at context path /examples
```

否则，将返回出错响应，它以 `FAIL` 开头，并包含一个出错消息，可能导致出误的原因包括：

- *Encountered exception*
  碰到异常情况，试图取消对某个 Web 应用的部署。可检查 Tomcat 日志了解详情。
- *Invalid context path was specified*
  上下文路径必须以斜杠字符开始，引用 ROOT Web 应用必须使用反斜杠（`/`）。
- *No context exists for path /foo* 在指定的上下文路径处没有部署的应用。
- *No context path was specified*
  需要指定 `path` 参数。

寻找内存泄露

`http://localhost:8080/manager/text/findleaks[?statusLine=[true|false]]`

寻找内存泄露的诊断将触发一个彻底的垃圾回收（**GC**）方案，所以如果在生产环境中使用它，需要非常谨慎才行。

寻找内存泄露的诊断会试图确认已导致内存泄露的 Web 应用（当其处于停止、重新加载，以及被取消部署状态时）。通常由一种分析器来确认结论。诊断使用了由 StandardHost（标准主机）实现所提供的附加功能。如果使用的是没有扩展自 StandHost 的自定义主机，则该诊断无法生效。

已有一些文档介绍，从 Java 代码中显式地触发彻底的垃圾回收方案是不可靠的。此外，在不同的 JVM 中，也有很多选项禁止显式触发垃圾回收，比如像 `-XX:+DisableExplicitGC`。 如果你需要确认诊断是否成功地实现了彻底的垃圾回收，可以使用 GC 日志、JConsole 分析器，或其他类似工具。

如果该命令成功执行，将得到类似如下的响应：

```
/leaking-webapp
```

如果你希望在响应中看到状态行，那么可以在请求中加入 `statusLine` 查询参数，并将其设定为 `true`。

对于已停止运行、被重新加载或被取消部署的Web 应用，由于之前运行所用到的类可能仍然加载在内存中，从而会造成内存泄露。响应将把这种应用的每个上下文路径都单列一行。如果应用被重新加载了数次，就可能会列出几次。

如果命令并没有成功执行，响应将以 `FAIL` 开头，并包含一个错误消息。

连接器 SSL/TLS 诊断

`http://localhost:8080/manager/text/sslConnectorCiphers`

SSL 连接器/加密诊断会列出当前每一连接器所配置的 SSL/TLS 加密算法。对于 BIO 和 NIO，将列出每个加密算法套件的名称；对于 APR，则返回 SSLCipherSuite 的值。

响应类似如下所示：

```
OK - Connector / SSL Cipher information
Connector[HTTP/1.1-8080]
  SSL is not enabled for this connector
Connector[HTTP/1.1-8443]
  TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
  TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
  ...
```

线程转储

`http://localhost:8080/manager/text/threaddump`

编写 JVM 线程转储。

响应类似如下所示：

```
OK - JVM thread dump
2014-12-08 07:24:40.080
Full thread dump Java HotSpot(TM) Client VM (25.25-b02 mixed mode):

"http-nio-8080-exec-2" Id=26 cpu=46800300 ns usr=46800300 ns blocked 0 for -1 ms
waited 0 for -1 ms
   java.lang.Thread.State: RUNNABLE
        locks java.util.concurrent.ThreadPoolExecutor$Worker@1738ad4
        at sun.management.ThreadImpl.dumpThreads0(Native Method)
        at sun.management.ThreadImpl.dumpAllThreads(ThreadImpl.java:446)
        at org.apache.tomcat.util.Diagnostics.getThreadDump(Diagnostics.java:440)
        at org.apache.tomcat.util.Diagnostics.getThreadDump(Diagnostics.java:409)
        at
org.apache.catalina.manager.ManagerServlet.threadDump(ManagerServlet.java:557)
        at
org.apache.catalina.manager.ManagerServlet.doGet(ManagerServlet.java:371)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:618)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:725)
...
```

虚拟机（VM）相关信息

`http://localhost:8080/manager/text/vminfo`

写入一些关于 Java 虚拟机（JVM）的诊断信息。

响应类似如下所示：

```
OK - VM info
2014-12-08 07:27:32.578
Runtime information:
  vmName: Java HotSpot(TM) Client VM
  vmVersion: 25.25-b02
  vmVendor: Oracle Corporation
  specName: Java Virtual Machine Specification
  specVersion: 1.8
  specVendor: Oracle Corporation
  managementSpecVersion: 1.2
  name: ...
  startTime: 1418012458849
  uptime: 393855
  isBootClassPathSupported: true

OS information:
...
```

保存配置信息

`http://localhost:8080/manager/text/save`

如果不指定任何参数，该命令将把服务器的当前配置信息保存到 server.xml 中。已有的配置信息 .xml 文件将被重命名，作为必要时的备份文件。

如果指定了 `path` 参数，而且该参数与已部署应用的路径相匹配，那么该 Web 应用的配置将保存为一个命名恰当的上下文 .xml 文件中，位于当前主机的 `xmlBase` 中。

要想使用该命令，则 StoreConfig MBean 必须存在。通常需要用 StoreConfigLifecycleListener来配置。

如果命令不能成功执行，响应将以 `FAIL` 开头，并包含一个错误消息。

服务器状态

可从下面这些链接中观察有关服务器的状态信息。任何一个 **manager-**** 角色都能访问这一页面。

```
http://localhost:8080/manager/status
http://localhost:8080/manager/status/all
```

上面是用 HTML 格式显示服务器状态信息的命令。

```
http://localhost:8080/manager/status?XML=true
http://localhost:8080/manager/status/all?XML=true
```

上面是用 XML 格式显示服务器状态信息的命令。

首先，显示的是服务器和 JVM 的版本号、JVM 提供者、操作系统的名称及其版本号，然后还显示了系统体系架构类型。

其次，显示的是关于 JVM 的内存使用信息。

最后，显示的是关于 Tomcat AJP 和 HTTP 连接器的信息。对两者来说，这些信息都很有用：

- 线程信息：最大线程数、最少及最多的空闲线程数、当前线程数量以及当前繁忙线程。
- 请求信息：最长及最短的处理时间、请求和错误的数量，以及接受和发送的字节数量。
- 一张完整显示线程阶段、时间、发送字节数、接受字节数、客户端、虚拟主机及请求的表。它将列出所有现有线程。下面列出了所有可能的线程阶段：
    - 解析及准备请求 将对请求报头进行解析，或进行必要的准备，以便读取请求主体（如果指定了传输编码）。
    - 服务 线程处理请求并生成响应。该阶段中至少有一个线程（可查看服务器状态页）。
    - 完成 请求处理结束。所有仍在输出缓冲区中的剩余响应都被传送至客户端。如果有必要保持连接活跃，则下一个阶段是“持续活跃”阶段，否则接下来直接进入“就绪”阶段。
    - 持续活跃 当客户端发送另一请求时，线程能使连接对客户端保持开放。如果接收到另一请求，下一阶段就将是“解析及准备请求”阶段。如果持续活跃超时结束，仍没有接收到请求，则连接关闭，进入下一阶段“就绪”阶段。
    - 就绪 线程空闲，等待再此被使用。

使用 `/status/all` 命令可查看每一个已配置 Web 应用的额外信息。

使用 JMX 代理 Servlet

什么是 JMX 代理 Servlet

JMX 代理 Servlet 是一款轻量级的代理。它的用途对用户来说并不是特别友好，但是其 UI 却非常有助于整合命令行脚本，从便于监控和改变 Tomcat 的内部运行。通过这个代理，我们可以获取和设置信息。要想真正了解 JMX 代理 Servlet，首先应该大概了解 JMX。如果不知道 JMX 的基本原理，那有些内容就很难理解了。

JMX 查询命令

JMX 的查询命令格式如下所示：

```
http://webserver/manager/jmxproxy/?qry=STUFF
```

`STUFF` 是所要执行的 JMX 查询。比如，可以执行以下这些查询：

- `qry=*%3Atype%3DRequestProcessor%2C* --> type=RequestProcessor` 定位所有能够处理请求并汇报各自状态的 Worker。
- `qry=*%3Aj2eeType=Servlet%2c* --> j2eeType=Servlet` 查询返回所有加载的 Servlet。
- `qry=Catalina%3Atype%3DEnvironment%2Cresourcetype%3DGlobal%2Cname%3DsimpleValue --> Catalina:type=Environment,resourcetype=Global,name=simpleValue` 按照指定名称查找 MBean。

需要实际地试验一下才能真正理解这些功能。如果没有提供 `qry` 参数，则将显示全部的 MBean。我们强烈建议你去阅读 Tomcat 源代码，真正了解 JMX 规范，更好地掌握所有能够执行的查询。

JMX 的 get 命令

JMXProxyServlet 还支持一种 `get` 命令来获取特定 MBean的属性值。该命令的一般格式如下所示：

```
http://webserver/manager/jmxproxy/?get=BEANNAME&att=MYATTRIBUTE&key=MYKEY
```

必须提供如下参数：

1. `get`：MBean 的完整名称。
2. `att`：希望获取的属性。
3. `key`：（可选参数）CompositeData MBean 的属性中的键。

如果命令成功执行，则一切正常，否则就会返回一个出错消息。举两个例子，比如当希望获取当前的堆内存数据时，可以采用如下命令：

```
http://webserver/manager/jmxproxy/?get=java.lang:type=Memory&att=HeapMemoryUsage
```

再或者，如果只希望获取“用过的”键，可以采用如下命令：

```
http://webserver/manager/jmxproxy/?
get=java.lang:type=Memory&att=HeapMemoryUsage&key=used
```

JMX 的 set 命令

上面介绍了如何查询一个 MBean。下面来看看 Tomcat 的内部运行吧！`set` 命令的一般格式为：

```
http://webserver/manager/jmxproxy/?set=BEANNAME&att=MYATTRIBUTE&val=NEWVALUE
```

需要提供三个请求参数：

- `set`：完整的 bean 名称。
- `att`：想要改变的属性。
- `val`：新的属性值。

如果命令成功执行，则一切正常，否则就会返回一个出错消息。比如，假如想为 `ErrorReportValve` 进行立即调试，可以将属性 `debug` 设为 10：

```
http://localhost:8080/manager/jmxproxy/
 ?set=Catalina%3Atype%3DValve%2Cname%3DErrorReportValve%2Chost%3Dlocalhost
 &att=debug&val=10
```

所得结果如下（你的有可能不同）：

```
Result: ok
```

下面来看看如果传入一个不恰当数值时的情况，比如使用一个URL，并试图将属性 debug 设置为 'cow'。

```
http://localhost:8080/manager/jmxproxy/
 ?set=Catalina%3Atype%3DValve%2Cname%3DErrorReportValve%2Chost%3Dlocalhost
 &att=debug&val=cow
```

运行结果如下：

```
Error: java.lang.NumberFormatException: For input string: "cow"
```

JMX 的 `invoke` 命令

使用 `invoke` 命令，我们就可以在 MBean 中调用方法。该命令的一般格式为：

```
http://webserver/manager/jmxproxy/
 ?invoke=BEANNAME&op=METHODNAME&ps=COMMASEPARATEDPARAMETERS
```

比如，使用如下方式来调用 **Service** 的 `findConnectors()` 方法：

```
http://localhost:8080/manager/jmxproxy/
 ?invoke=Catalina%3Atype%3DService&op=findConnectors&ps=
```

利用 Ant 执行 Manager 的命令

上面的文档介绍了如何利用 HTTP 请求来执行 Manager 的命令。除此之外，Tomcat 还专为 Ant（1.4 版或更新版本）构建工具准备了一套方便的任务定义。为了使用这些命令，必须执行下面这些操作：

- 下载 Ant 二进制分发包，地址为：http://ant.apache.org。必须使用 1.4 版本或更新版本。
- 将分发包安装到合适的目录中（下面将把它叫做 ANT_HOME）。
- 将文件 `server/lib/catalina-ant.jar` 从 Tomcat 安装目录中复制到 Ant 的库目录（`$ANT_HOME/lib`）。
- 将 `$ANT_HOME/bin` 目录添加到环境变量 `PATH` 中。
- 在 Tomcat 用户数据库中，至少配置一个拥有 `manager-script` 角色的用户名/密码组合数据。

为了在 Ant 中使用自定义任务，必须首先用 `<taskdef>` 元素来声明它们，因而 `build.xml` 文件应类似如下这样：

```
<project name="My Application" default="compile" basedir=".">

  <!-- Configure the directory into which the web application is built -->
  <property name="build"    value="${basedir}/build"/>

  <!-- Configure the context path for this application -->
  <property name="path"     value="/myapp"/>

  <!-- Configure properties to access the Manager application -->
  <property name="url"      value="http://localhost:8080/manager/text"/>
  <property name="username" value="myusername"/>
  <property name="password" value="mypassword"/>

  <!-- Configure the custom Ant tasks for the Manager application -->
  <taskdef name="list"      classname="org.apache.catalina.ant.ListTask"/>
  <taskdef name="deploy"    classname="org.apache.catalina.ant.DeployTask"/>
  <taskdef name="start"     classname="org.apache.catalina.ant.StartTask"/>
  <taskdef name="reload"    classname="org.apache.catalina.ant.ReloadTask"/>
  <taskdef name="stop"      classname="org.apache.catalina.ant.StopTask"/>
  <taskdef name="undeploy"  classname="org.apache.catalina.ant.UndeployTask"/>
  <taskdef name="resources" classname="org.apache.catalina.ant.ResourcesTask"/>
  <typedef name="sessions"  classname="org.apache.catalina.ant.SessionsTask"/>
  <taskdef name="findleaks" classname="org.apache.catalina.ant.FindLeaksTask"/>
  <typedef name="vminfo"    classname="org.apache.catalina.ant.VminfoTask"/>
  <typedef name="threaddump" classname="org.apache.catalina.ant.ThreaddumpTask"/>
  <typedef name="sslConnectorCiphers"
classname="org.apache.catalina.ant.SslConnectorCiphersTask"/>

  <!-- Executable Targets -->
  <target name="compile" description="Compile web application">
    <!-- ... construct web application in ${build} subdirectory, and
         generated a ${path}.war ... -->
  </target>

  <target name="deploy" description="Install web application"
          depends="compile">
    <deploy url="${url}" username="${username}" password="${password}"
            path="${path}" war="file:${build}${path}.war"/>
  </target>

  <target name="reload" description="Reload web application"
          depends="compile">
```

```
    <reload  url="${url}" username="${username}" password="${password}"
            path="${path}"/>
  </target>

  <target name="undeploy" description="Remove web application">
    <undeploy url="${url}" username="${username}" password="${password}"
            path="${path}"/>
  </target>

</project>
```

注意：上面的资源任务定义将覆盖 Ant 1.7 中所添加的资源数据类型。如果你希望使用这些资源数据类型，需要使用 Ant 命名空间支持，将 Tomcat 的任务分配到它们自己的命名空间中。

现在，可以执行类似 `ant deploy` 这样的命令将应用部署到 Tomcat 的一个运行实例上，或者利用 `ant reload` 通知 Tomcat 重新加载应用。另外还需注意的是，在这个 `build.xml` 文件中，多数比较有价值的属性值都是可以被可替换的，因而可以利用命令行方式来重写这些值。比如，考虑到在 `build.xml` 文件中包含真正的管理员密码是非常危险的，可以通过一些命令来忽略密码属性，如下所示：

```
ant -Dpassword=secret deploy
```

任务输出捕获

使用 Ant **1.6.2** 版或更新版本，Catalina 任务提供选项，利用属性或外部文件捕获输出。它们直接支持 `<redirector>` 类型属性的子集：

属性	属性说明	是否必需
`output`	输出文件名。如果错误流没有重定向到一个文件或属性上，它将出现在输出中。	否
`error`	命令的标准错误应该被重定向到的文件。	否
`logError`	用于在 Ant 日志中显示错误输出，将输出重定向至某个文件或属性。错误输出不会包含在输出文件或属性中。如果利用 `error` 或 `errorProperty` 属性重定向错误，则没有任何效果。	否
`append`	输出和错误文件是否应该附加或覆盖。默认为 `false`。	否

createemptyfiles	是否应该创建输出和错误文件，哪怕是空的文件。默认为 true。	否
outputproperty	用于保存命令输出的属性名。除非错误流被重定向至单独的文件或流，否则这一属性将包含错误输出。	否
errorproperty	用于保存命令标准错误的属性名。	否

还可以指定其他一些额外属性：

属性	属性说明	是否必需
alwaysLog	该属性用于查看捕获的输出，这个输出也出现在 Ant 日志中。除非捕获任务输出，否则千万不要使用它。默认为 false。**Ant 1.6.3** 通过 **<redirector>** 直接支持该属性。	否
failonerror	用于避免因为 manager 命令处理中错误而导致 Ant 执行终止情况的发生。默认为 true。如果希望捕获错误输出，则必须设为 false，否则 Ant 执行将有可能在未捕获任何输出前就被终止。该属性只用于 manager 命令的执行上，任何错误的或丢失的命令属性仍然会导致 Ant 执行终止。	否

它们还支持内嵌的 <redirector> 元素，你可以在这些元素中指定全套的属性。但对于 input、inputstring、inputencoding，即使接收，也无法使用，因为在这种上下文中它们没有任何意义。详情可参考 Ant 手册以了解 <redirector> 元素的各个属性。

下面这个范例摘录了一段构建文件，展示了这种对输出重定向的支持是如何运作的。

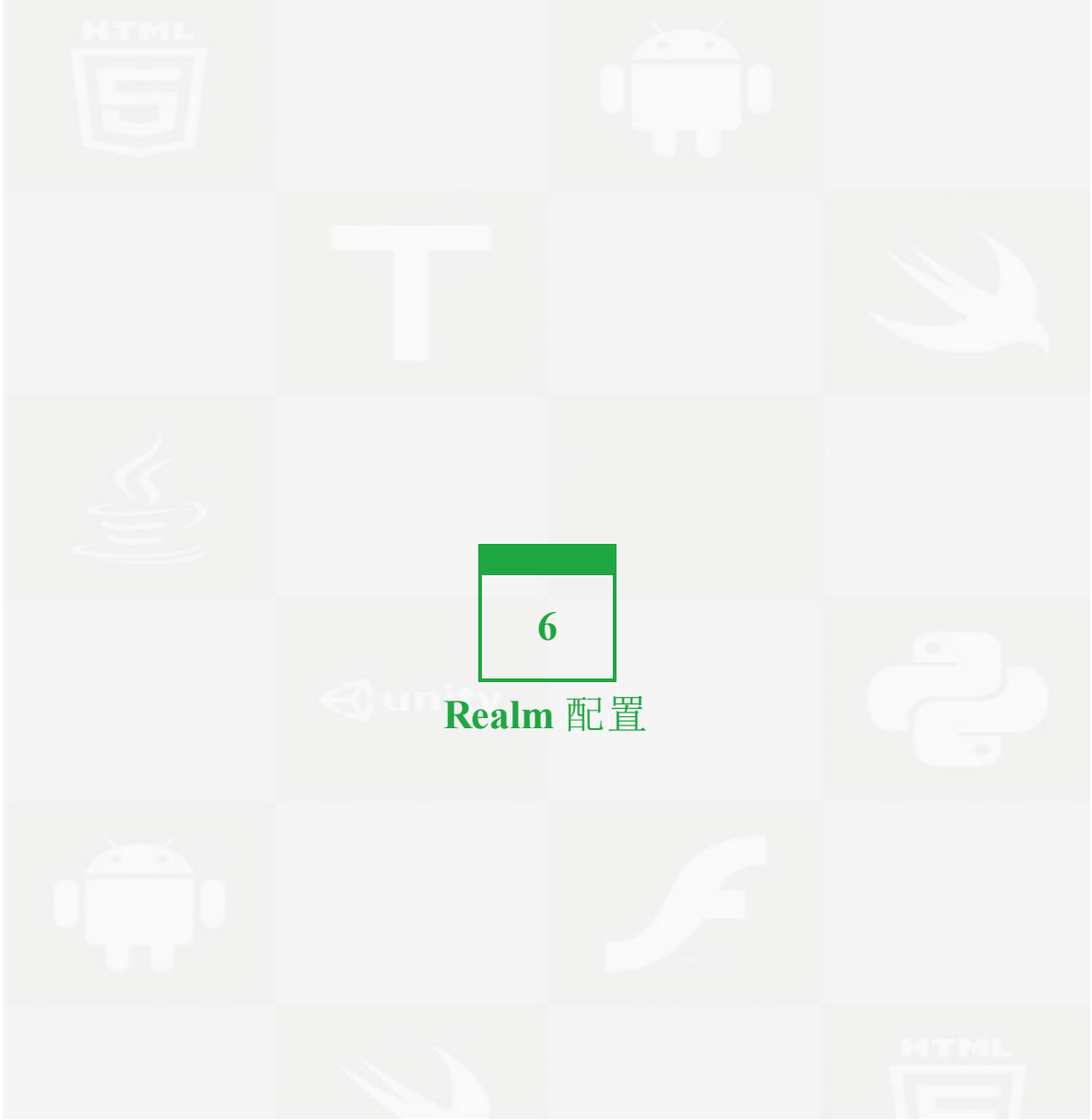
```
    <target name="manager.deploy"
        depends="context.status"
        if="context.notInstalled">
        <deploy url="${mgr.url}"
            username="${mgr.username}"
            password="${mgr.password}"
            path="${mgr.context.path}"
            config="${mgr.context.descriptor}"/>
```

```
    </target>

    <target name="manager.deploy.war"
        depends="context.status"
        if="context.deployable">
        <deploy url="${mgr.url}"
            username="${mgr.username}"
            password="${mgr.password}"
            update="${mgr.update}"
            path="${mgr.context.path}"
            war="${mgr.war.file}"/>
    </target>

    <target name="context.status">
        <property name="running" value="${mgr.context.path}:running"/>
        <property name="stopped" value="${mgr.context.path}:stopped"/>

        <list url="${mgr.url}"
            outputproperty="ctx.status"
            username="${mgr.username}"
            password="${mgr.password}">
        </list>

        <condition property="context.running">
            <contains string="${ctx.status}" substring="${running}"/>
        </condition>
        <condition property="context.stopped">
            <contains string="${ctx.status}" substring="${stopped}"/>
        </condition>
        <condition property="context.notInstalled">
            <and>
                <isfalse value="${context.running}"/>
                <isfalse value="${context.stopped}"/>
            </and>
        </condition>
        <condition property="context.deployable">
            <or>
                <istrue value="${context.notInstalled}"/>
                <and>
                    <istrue value="${context.running}"/>
                    <istrue value="${mgr.update}"/>
                </and>
                <and>
                    <istrue value="${context.stopped}"/>
                    <istrue value="${mgr.update}"/>
                </and>
            </or>
        </condition>
        <condition property="context.undeployable">
            <or>
                <istrue value="${context.running}"/>
                <istrue value="${context.stopped}"/>
            </or>
        </condition>
    </target>
```

警告：多次调用 Catalina 任务往往并不是一个好主意，退一步说这样做的意义也不是很大。如果 Ant 任务依赖链设定糟糕的话，即使本意并非如此，也会导致在一次 Ant 运行中多次运行任务。必须提前对你稍加警告，因为有可能当你从任务中捕获输出时，会出现一些意想不到的情况：

- 当用属性捕获时，你将只能从其中找到最初调用的输出，因为 Ant 属性是不变的，一旦设定就无法改变。

- 当用文件捕获时，你将只能从其中找到最后调用的输出，除非使用 `append = "true"` 属性——在这种情况下，你将看到附加在文件内容末尾的每一个任务调用的相关输出。



6

Realm 配置

快速入门

本文档介绍了如何借助一个“数据库”来配置 Tomcat ，从而实现容器管理安全性。所要连接的这种数据库含有用户名、密码以及用户角色。你只需知道的是，如果使用的 Web 应用含有一个或多个 `<security-constraint>` 元素，`<login-config>` 元素定义了用户验证的必需细节信息。如果你不打算使用这些功能，则可以忽略这篇文档。

关于容器管理安全性的基础知识，可参考 Servlet Specification (Version 2.4) 中的第 12 节内容。

关于如何使用 Tomcat 中的单点登录（用户只需验证一次，就能够登录一个虚拟主机的所有 Web 应用）功能，请参看该文档。

概述

什么是 Realm

Realm（安全域）其实就是一个存储用户名和密码的“数据库”再加上一个枚举列表。“数据库”中的用户名和密码是用来验证 Web 应用（或 Web 应用集合）用户合法性的，而每一合法用户所对应的角色存储在枚举列表中。可以把这些角色看成是类似 UNIX 系统中的 group（分组），因为只有能够拥有特定角色的用户才能访问特定的 Web 应用资源（而不是通过对用户名列表进行枚举适配）。特定用户的用户名下可以配置多个角色。

虽然 Servlet 规范描述了一个可移植机制，使应用可以在 `web.xml` 部署描述符中声明它们的安全需求，但却没有提供一种可移植 API 来定义出 Servlet 容器与相应用户及角色信息的接口。然而，在很多情况下，非常适于将 Servlet 容器与一些已有的验证数据库或者生产环境中已存在的机制“连接”起来。因此，Tomcat 定义了一个 Java 接口（`org.apache.catalina.Realm`），通过“插入”组件来建立连接。提供了 6 种标准插件，支持与各种验证信息源的连接：

- JDBCRealm——通过 JDBC 驱动器来访问保存在关系型数据库中的验证信息。
- DataSourceRealm——访问保存在关系型数据库中的验证信息。
- JNDIRealm——访问保存在 LDAP 目录服务器中的验证信息。
- UserDatabaseRealm——访问存储在一个 UserDatabase JNDI 数据源中的认证信息，通常依赖一个 XML 文档（`conf/tomcat-users.xml`）。
- MemoryRealm——访问存储在一个内存中对象集合中的认证信息，通过 XML 文档初始化（`conf/tomcat-users.xml`）。
- JAASRealm——通过 Java 认证与授权服务（JAAS）架构来获取认证信息。

另外，还可以编写自定义 `Realm` 实现，将其整合到 Tomcat 中，只需这样做：

- 实现 `org.apache.catalina.Realm` 接口。
- 将编译好的 realm 放到 $CATALINA_HOME/lib 中。
- 声明自定义 realm，具体方法详见“配置 Realm”一节。
- 在 MBeans 描述符文件中声明自定义realm。

配置 Realm

在详细介绍标准 Realm 实现之前，简要了解 Realm 的配置方式是很关键的一步。大体来说，就是需要在 `conf/server.xml` 配置文件中添加一个 XML 元素，如下所示：

```
<Realm className="... class name for this implementation"
       ... other attributes for this implementation .../>
```

`<Realm>` 可以嵌入以下任何一种 `Container` 元素中。Realm 元素的位置至关重要，它会对 Realm 的“范围”（比如说哪个 Web 应用能够共享同一验证信息）有直接的影响。

- **`<Engine>`** 元素 内嵌入该元素中的这种 Realm 元素可以被所有虚拟主机上的所有 Web 应用所共享，除非该 Realm 元素被内嵌入下属 `<Host>` 或 `<Context>` 元素的 Realm 元素所覆盖。
- **`<Host>`** 元素 内嵌入该元素中的这种 Realm 元素可以被这一虚拟主机上的所有 Web 应用所共享。除非该 Realm 元素被内嵌入下属 `<Context>` 元素的 Realm 元素所覆盖。
- **`<Context>`** 元素 内嵌入该元素中的这种 Realm 元素只能被这一 Web 应用所使用。

常用特性

摘要式密码

对于每种标准 `Realm` 实现来说，用户的密码默认都是以明文方式保存的。在很多情况下，这种方式都非常糟糕，即使是一般的用户也能收集到足够的验证信息，从而以其他用户的信息成功登录。为了避免这种情况的发生，标准 `Realm` 实现支持一种对用户密码进行摘要式处理的机制，它能以无法轻易破解的形式对存储的密码进行加密处理，同时保证`Realm` 实现仍能使用这种加密后的密码进行验证。

在标准的 Realm 验证时，会将存储的密码与用户所提供的密码进行比对，这时，我们可以通过指定 `<Realm>` 元素中的 `digest` 属性来选择摘要式密码。该属性值必须是一种`java.security.MessageDigest` 类所支持的摘要式算法（SHA、MD2、或 MD5）。当你选择该属性值时，存储在 Realm 中的密码内容必须是明文格式，随后它将被你所指定的算法进行摘要式加密。

在调用 Realm 的 `authenticate()` 方法后，用户所提供的明文密码同样也会利用上述你所指定的加密算法进行加密，加密结果与 Realm 的返回值相比较。如果两者相等，则表明原始密码的明文形式更用户所提供的密码完全等同，因此该用户身份验证成功。

可以采用以下两种比较便利的方法来计算明文密码的摘要值：

- 如果应用需要动态计算摘要式密码，调用 `org.apache.catalina.realm.RealmBase` 类的静态 `Digest()` 方法，传入明文密码和摘要式算法名称及字符编码方案。该方法返回摘要式密码。
- 如果想执行命令行工具来计算摘要式密码，只需执行：
  `CATALINA_HOME/bin/digest.[bat|sh] -a {algorithm} {cleartext-password}`
  标准输出将返回明文密码的摘要式形式。

如果使用 DIGEST 验证的摘要式密码，用来生成摘要密码的明文密码则将有所不同，而且必须使用一次不加盐的 MD5 算法。对应到上面的范例，那就是必须把 `{cleartext-password}` 替换成 `{username}:{realm}:{cleartext-password}`。再比如说，在一个开发环境中，可能采用这种形式：`testUser:Authentication required:testPassword`。`{realm}` 的值取自 Web 应用 `<login-config>` 的 `<realm-name>` 元素。如果没有在 web.xml 中指定，则使用默认的`Authentication required`。

若要使用非平台默认编码的用户名和（或）密码，则命令如下：

```
CATALINA_HOME/bin/digest.[bat|sh] -a {algorithm} -e {encoding} {input}
```

但需要注意的是，一定要确保输入正确地传入摘要。摘要返回 `{input}:{digest}`。如果输入在返回时出现损坏，摘要则将无效。

摘要的输出格式为 `{salt}${iterations}${digest}`。如果盐的长度为 0，迭代次数为 1，则输出将简化为 `{digest}`。

`CATALINA_HOME/bin/digest.[bat|sh]` 的完整格式如下：

```
CATALINA_HOME/bin/digest.[bat|sh] [-a <algorithm>] [-e <encoding>]
        [-i <iterations>] [-s <salt-length>] [-k <key-length>]
        [-h <handler-class-name>] <credentials>
```

- `-a` 用来生成存储凭证的算法。如未指定，将使用凭证处理器（CredentialHandler）的默认值，如果认证处理器和算法均未指定，则使用默认值 `SHA-512`。
- `-e` 指定用于任何必要的字节与字符间转换的字符编码方案。如未指定，使用系统默认的字符编码方案 `Charset#defaultCharset()`。
- `-i` 生成存储的凭证时所使用的迭代次数。如未指定，使用 CredentialHandler 的默认值。
- `-s` 生成并存储到认证中的盐的长度（字节）。如未指定，使用 CredentialHandler 的默认值。
- `-k` （生成凭证时，如果随带创建了）键的长度（位）。如未指定，则使用 CredentialHandler 的默认值

- `-h` CredentialHandler 使用的完整类名。如未指定，则轮流使用内建的凭证处理器（`MessageDigestCredentialHandler`，然后是 `SecretKeyCredentialHandler`），将使用第一个接受指定算法的凭证处理器。

范例应用

Tomcat 自带的范例应用中包含一个受到安全限制保护的区域，使用表单式登录方式。为了访问它，在你的浏览器地址栏中输入 `http://localhost:8080/examples/jsp/security/protected/`，并使用 UserDatabaseRealm 默认的用户名和密码进行登录。

Manager 应用

如果你希望使用 Manager 应用在一个运行的 Tomcat 安装上来部署或取消部署 Web 应用，那么必须在一个选定的 Realm 实现上，将 **manager-gui** 角色添加到至少一个用户名上。这是因为 Manager 自身使用一个安全限制，要想在该应用的 HTML 界面中访问请求 URI，就必须要有 manager-gui 角色。

出于安全性考虑，默认情况下，Realm 中的用户名（比如使用 `conf/tomcat-users.xml`）没有被分配 manager-gui 角色。因此，用户起初无法使用这个功能，除非 Tomcat 管理员特意将这一角色分配给他们。

Realm 日志

Realm 的容器（Context、Host 及 Engine）所对应的日志配置文件将记录 `Realm` 所记录下的调试和异常信息。

标准 Realm 实现

JDBCRealm

简介

JDBCRealm 是 Tomcat `Realm` 接口的一种实现，它通过 JDBC 驱动程序在关系型数据库中查找用户。只要数据库结构符合下列要求，你可以通过大量的配置来灵活地修改现有的表与列名。

- 必须有一张用户表（*users table*）。它包含着一个由 `Realm` 所能识别的所有合法用户所构成的行。
- 用户表必须至少包含两列（当然，如果现有应用确实需要，则同样也可以包含更多的列）：
  - 用户名。当用户登录时，能被 Tomcat 识别的用户名。
  - 密码。当用户登录时，能被 Tomcat 所识别的密码。该列中的值可能是明文，也可能是摘要式密码，稍后详述。
- 必须有一张用户角色表（*user roles table*）。该表包含一个角色行，包含着可能指定给特定用户的每个合法角色。一个用户可以没有角色，也可以有一个或多个角色，这都是合法的。
- 用户角色表 至少应包含两列（如果现有应用确实需要，则也可以包含更多的列）：
  - 用户名。Tomcat 所能识别的用户名（与用户表中指定的值相同）。
  - 用户所对应的合法角色名。

快速入门

为了设置 Tomcat 从而使用 JDBCRealm，需要执行以下步骤：

1. 在数据库中创建符合上述规范的表与列。
2. 配置一个 Tomcat 使用的数据库用户名与密码，并且至少有只读权限（Tomcat 永远都不会去修改那些表中的数据）。
3. 将用到的 JDBC 驱动程序复制到 `$CATALINA_HOME/lib` 目录中。注意只能识别 JAR 文件！
4. 在 `$CATALINA_BASE/conf/server.xml` 目录中设置一个 `<Realm>` 元素。这一点下文将会详细叙述。
5. 如果 Tomcat 处于运行状态，则重启它。

Realm 元素属性

如上所述，为了配置 JDBCRealm，需要创建一个 `Realm` 元素，并把它放在 `$CATALINA_BASE/conf/server.xml` 文件中。JDBCRealm 的属性都定义在 Realm 配置文档中。

范例

下面这个 SQL 脚本范例创建了我们所需的表（根据你所用的数据库，可以相应修改其中的语法）。

```
create table users (
  user_name         varchar(15) not null primary key,
  user_pass         varchar(15) not null
);

create table user_roles (
  user_name         varchar(15) not null,
```

```
  role_name         varchar(15) not null,
  primary key (user_name, role_name)
);
```

`Realm` 元素包含在默认的 `$CATALINA_BASE/conf/server.xml` 文件中（被注释掉了）。在下面的范例中，有一个名为 authority 的数据库，它包含上述创建的表，通过用户名“dbuser”和密码“dbpass”进行访问。

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
      driverName="org.gjt.mm.mysql.Driver"
   connectionURL="jdbc:mysql://localhost/authority?user=dbuser&amp;password=dbpass"
       userTable="users" userNameCol="user_name" userCredCol="user_pass"
   userRoleTable="user_roles" roleNameCol="role_name"/>
```

特别注意事项

JDBCRealm 必须遵循以下规则：

- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法，从而使任何对数据库的即时修改（新用户、密码或角色改变，等等）都能立即生效。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。
- 应用负责管理**users**（用户表）和**user roles**（用户角色表）中的信息。Tomcat 没有提供任何内置功能来维护这两种表。

DataSourceRealm

简介

DataSourceRealm 是 Tomcat `Realm` 接口的一种实现，它通过一个 JNDI 命名的 JDBC 数据源在关系型数据库中查找用户。只要数据库结构符合下列要求，你可以通过大量的配置来灵活地修改现有的表与列名。

- 必须有一张用户表（*users table*）。它包含着一个由 `Realm` 所能识别的所有合法用户所构成的行。
- 用户表必须至少包含两列（当然，如果现有应用确实需要，则同样也可以包含更多的列）：
  - 用户名。当用户登录时，能被 Tomcat 识别的用户名。
  - 密码。当用户登录时，能被 Tomcat 所识别的密码。该列中的值可能是明文，也可能是摘要式密码，稍后详述。
- 必须有一张用户角色表（*user roles table*）。该表包含一个角色行，包含着可能指定给特定用户的每个合法角色。一个用户可以没有角色，也可以有一个或多个角色，这都是合法的。
- 用户角色表 至少应包含两列（如果现有应用确实需要，则也可以包含更多的列）：
  - 用户名。Tomcat 所能识别的用户名（与用户表中指定的值相同）。
  - 用户所对应的合法角色名。

快速入门

为了设置 Tomcat 从而使用 DataSourceRealm，需要执行以下步骤：

1. 在数据库中创建符合上述规范的表与列。
2. 配置一个 Tomcat 使用的数据库用户名与密码，并且至少有只读权限（Tomcat 永远都不会去修改那些表中的数据）。

3. 为数据库配置一个 JNDI 命名的 JDBC DataSource。详情可参考JNDI DataSource Example HOW-TO》应该链接至相应中文页面》。
4. 在 `$CATALINA_BASE/conf/server.xml` 目录中设置一个 `<Realm>` 元素。这一点下文将会详细叙述。
5. 如果 Tomcat 处于运行状态，则重启它。

范例

下面这个 SQL 脚本范例创建了我们所需的表（根据你所用的数据库，可以相应修改其中的语法）。

```
create table users (
  user_name         varchar(15) not null primary key,
  user_pass         varchar(15) not null
);

create table user_roles (
  user_name         varchar(15) not null,
  role_name         varchar(15) not null,
  primary key (user_name, role_name)
);
```

在下面的范例中，有一个名为 authority 的 MySQL 数据库，它包含上述创建的表，通过名为“`java:/comp/env/jdbc/authority`”的 JNDI 命名的 JDBC 数据源来访问。

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
   dataSourceName="jdbc/authority"
   userTable="users" userNameCol="user_name" userCredCol="user_pass"
   userRoleTable="user_roles" roleNameCol="role_name"/>
```

特别注意事项

使用 DataSourceRealm 时必须遵守下列规则：

- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法，从而使任何对数据库的即时修改（新用户、密码或角色改变，等等）都能立即生效。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。
- 应用负责管理**users**（用户表）和**user roles**（用户角色表）中的信息。Tomcat 没有提供任何内置功能来维护这两种表。

JNDIRealm

简介

JNDIRealm 是 Tomcat `Realm` 接口的一种实现，通过一个 JNDI 提供者[1]在 LDAP 目录服务器中查找用户。realm 支持大量的方法来使用认证目录。

通常是可以使用 JNDI API 类的标准 LDAP 提供者。

a. 连接目录

realm 与目录服务器的连接是通过 **connectionURL** 配置属性来定义的。这个 URL 的格式是通过 JNDI 提供者来定义的。它通常是个 LDAP URL，指定了所要连接的目录服务器的域名，另外还（可选择）指定了所需的根命名上下文的端口和唯一名称（DN）。

如果有多个提供者，则可以配置 **alternateURL**。如果一个套接字连接无法传递给提供者 **connectionURL**，则会换用 **alternateURL**。

当通过创建连接来搜索目录，获取用户及角色信息时，realm 会利用 **connectionName** 和 **connectionPassword** 这两个属性所指定的用户名和密码在目录上进行自我认证。如果未指定这两个属性，则创立的连接是匿名连接，这种连接适用于大多数情况。

b. 选择用户目录项

在目录中，每个可被认证的用户都必须表示为独立的项，这种独立项对应着由属性 **connectionURL** 定义的初始 `DirContext` 中的元素。这种用户项必须有一个包含认证所需用户名的属性。

每个用户项的唯一性名称（DN）通常含有用于认证的用户名。在这种情况下，**userPattern** 属性可以用来指定 DN，其中的 `{0}` 代表用户名应该被替换的位置。

realm 必须搜索目录来寻找一个包含用户名的唯一项，可用下列属性来配置搜索：

- **userBase** 用户子树的基准项。如果未指定，则搜索基准为顶级元素。
- **userSubtree** 用户子树，也就是搜索范围。如果希望搜索以**userBase** 项为基准的整个子树，则使用 `true`；默认值为 `false`，只对顶级元素进行搜索。
- **userSearch** 指定替代用户名之后所使用的 LDAP 搜索过滤器的模式。

c. 用户认证

- 绑定模式

  默认情况下，realm 会利用用户项的 DN 与用户所提供的密码，将用户绑定到目录上。如果成功执行了这种简单的绑定，那么就可以认为用户认证成功。

  出于安全考虑，目录可能保存的是用户的摘要式密码，而非明文密码（参看摘要式密码以获知详情）。在这种情况下，在绑定过程中，目录会自动将用户所提供的明文密码加密为正确的摘要式密码，以便后续和存储的摘要式密码进行比对。然而在绑定过程中，realm 并不参与处理摘要式密码。不会用到 **digest** 属性，如果设置了该属性，也会被自动忽略。

- 对比模式

  另外一种方法是，realm 从目录中获取存储的密码，然后将其与用户所提供的值进行比对。配置方法是，在包含密码的用户项中，将 **userPassword** 属性设为目录属性名。

  对比模式的缺点在于：首先，`connectionName` 和 `connectionPassword` 属性必须配置成允许 realm 读取目录中的用户密码。出于安全考虑，这是一种不可取的做法。事实上，很多目录实现甚至都不允许目录管理器读取密码。其次，realm 必须自己处理摘要式密码，包括要设置所使用的具体算法、在目录中表示密码散列值的方式。但是，realm 可能有时又要访问存储的密码，比如为了支持 HTTP 摘要式访问认证（HTTP Digest Access Authentication，RFC 2069）。（注意，HTTP 摘要式访问认证不同于之前讨论过的在库中存储密码摘要的方式。）

d. 赋予用户角色

Realm 支持两种方法来表示目录中的角色：

- 将角色显式表示为目录项

  通过明确的目录项来表示角色。角色项通常是一个 LDAP 分组项，该分组项的一个属性包含角色名称，另一属性值则是拥有该角色的用户的 DN 名或用户名。下列属性配置了一个目录搜索来寻找与认证用户相关的角色名。

  - **roleBase** 角色搜索的基准项。如未指定，则基准项为顶级目录上下文。
  - **roleSubtree** 搜索范围。如果希望搜索以 `roleBase` 为基准项的整个子树，则设为 `true`。默认值为 `false`，请求一个只包含顶级元素的单一搜索。
  - **roleSearch** 用于选择角色项的 LDAP 搜索过滤器。它还可以（可选择）包含用于唯一名称的模式替换 `{0}`、用户名的模式替换 `{1}`，以及用户目录项属性的模式替换 `{2}`。使用 **userRoleAttribute** 来指定提供 `{2}` 值的属性名。

- **roleName** 包含角色名称的角色项属性。
- **roleNested** 启用内嵌角色。如果希望在角色中内嵌角色，则设为 `true`。如果配置了该属性，每一个新近找到的 roleName 和 DN 都将用于递归式的新角色搜索。默认值为 `false`。

- 将角色表示为用户项属性

  将角色名称保存为用户目录项中的一个属性值。使用 **userRoleName** 来指定该属性名称。

当然，也可以综合使用这两种方法来表示角色。

快速入门

为了配置 Tomcat 使用 JNDIRealm，需要下列步骤：

1. 确保目录服务器配置中的模式符合上文所列出的要求。
2. 必要时可以为 Tomcat 配置一个用户名和密码，对上文所提到的信息用于只读访问权限（Tomcat 永远不会修改该信息。）
3. 按照下文的方法，在 `$CATALINA_BASE/conf/server.xml` 文件中设置一个 `<Realm>` 元素。
4. 如果 Tomcat 已经运行，则重启它。

Realm 元素属性

如上所述，为了配置 JDBCRealm，需要创建一个 `Realm` 元素，并把它放在 `$CATALINA_BASE/conf/server.xml` 文件中。JDBCRealm 的属性都定义在 Realm 配置文档中。

范例

在目录服务器上创建适合的模式超出了本文档的讲解范围，因为这是跟每个目录服务器的实现密切相关的。在下面的实例中，我们将假定使用的是 OpenLDAP 目录服务器的一个分发版（2.0.11 版或更新版本，可从 http://www.openldap.org处下载）。假设 `slapd.conf` 文件包含下列设置（除了其他设置之外）。

```
database ldbm
suffix dc="mycompany",dc="com"
rootdn "cn=Manager,dc=mycompany,dc=com"
rootpw secret
```

我们还假定 `connectionURL`，使目录服务器与 Tomcat 运行在同一台机器上。要想了解如何配置及使用 JNDI LDAP 提供者的详细信息，请参看 http://docs.oracle.com/javase/7/docs/technotes/guides/jndi/index.html。

接下来，假定利用如下所示的元素（以 LDIF 格式）来填充目录服务器。

```
 # Define top-level entry

dn: dc=mycompany,dc=com
objectClass: dcObject
dc:mycompany

 # Define an entry to contain people

 # searches for users are based on this entry

dn: ou=people,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: people

 # Define a user entry for Janet Jones
```

```
dn: uid=jjones,ou=people,dc=mycompany,dc=com
objectClass: inetOrgPerson
uid: jjones
sn: jones
cn: janet jones
mail: j.jones@mycompany.com
userPassword: janet

 # Define a user entry for Fred Bloggs

dn: uid=fbloggs,ou=people,dc=mycompany,dc=com
objectClass: inetOrgPerson
uid: fbloggs
sn: bloggs
cn: fred bloggs
mail: f.bloggs@mycompany.com
userPassword: fred

 # Define an entry to contain LDAP groups

 # searches for roles are based on this entry

dn: ou=groups,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: groups

 # Define an entry for the "tomcat" role

dn: cn=tomcat,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: tomcat
uniqueMember: uid=jjones,ou=people,dc=mycompany,dc=com
uniqueMember: uid=fbloggs,ou=people,dc=mycompany,dc=com

 # Define an entry for the "role1" role

dn: cn=role1,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: role1
uniqueMember: uid=fbloggs,ou=people,dc=mycompany,dc=com
```

OpenLDAP 服务器。假定用户使用他们的 uid（比如说 jjones）登录应用，匿名连接已经足够可以搜索目录并获取角色信息了：

```
<Realm   className="org.apache.catalina.realm.JNDIRealm"
     connectionURL="ldap://localhost:389"
       userPattern="uid={0},ou=people,dc=mycompany,dc=com"
          roleBase="ou=groups,dc=mycompany,dc=com"
          roleName="cn"
        roleSearch="(uniqueMember={0})"
/>
```

利用这种配置，通过在 `userPattern` 替换用户名，realm 能够确定用户的 DN，然后利用这个 DN 和取自用户的密码将用户绑定到目录中，从而验证用户身份，然后搜索整个目录服务器来找寻用户角色。

现在假定希望用户输入电子邮件地址（而不是用户 id）。在这种情况下，realm 必须搜索目录找到用户项。当用户项被保存在多个子树中，而这些子树可能分别对应不同的组织单位或企业位置时，可能必须执行一个搜索。

另外，假设除了分组项之外，你还想用用户项的属性来保存角色，那么在这种情况下，Janet Jones 对应的项可能如下所示：

```
dn: uid=jjones,ou=people,dc=mycompany,dc=com
```

```
objectClass: inetOrgPerson
uid: jjones
sn: jones
cn: janet jones
mail: j.jones@mycompany.com
memberOf: role2
memberOf: role3
userPassword: janet
```

这个 realm 配置必须满足以下新要求：

```
<Realm   className="org.apache.catalina.realm.JNDIRealm"
     connectionURL="ldap://localhost:389"
          userBase="ou=people,dc=mycompany,dc=com"
        userSearch="(mail={0})"
      userRoleName="memberOf"
          roleBase="ou=groups,dc=mycompany,dc=com"
          roleName="cn"
        roleSearch="(uniqueMember={0})"
/>
```

当 Janet Jones 用她的电子邮件 `j.jones@mycompany.com` 登录时，realm 会搜索目录，寻找带有该电邮值的唯一项，并尝试利用给定密码来绑定到目录：`uid=jjones,ou=people,dc=mycompany,dc=com`。如果验证成功，该用户将被赋予以下三个角色："role2" 与 "role3"，她的目录项中的 `memberOf` 属性值："tomcat"，她作为成员存在的唯一分组项中的 `cn` 属性值。

最后，为了验证用户，我们必须从目录中获取密码并在 realm 中执行本地比对，将 realm 按照如下方式来配置：

```
<Realm   className="org.apache.catalina.realm.JNDIRealm"
    connectionName="cn=Manager,dc=mycompany,dc=com"
connectionPassword="secret"
     connectionURL="ldap://localhost:389"
      userPassword="userPassword"
       userPattern="uid={0},ou=people,dc=mycompany,dc=com"
          roleBase="ou=groups,dc=mycompany,dc=com"
          roleName="cn"
        roleSearch="(uniqueMember={0})"
/>
```

但是，正如之前所讨论的那样，往往应该优先考虑默认的绑定模式。

特别注意事项

使用 JNDIRealm 需要遵循以下规则：

- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法，从而使任何对数据库的即时修改（新用户、密码或角色改变，等等）都能立即生效。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。
- 应用负责管理**users**（用户表）和**user roles**（用户角色表）中的信息。Tomcat 没有提供任何内置功能来维护这两种表。

UserDatabaseRealm

UserDatabaseRealm 是 Tomcat **Realm** 接口的一种实现，使用 JNDI 资源来存储用户信息。默认，JNDI 资源是通过一个 XML 文件来提供支持的。它并不是针对大规模生产环境用途而设计的。在启动时，UserDatabaseRealm 会

从一个 XML 文档中加载所有用户以及他们角色的信息（该 XML 文档默认位于 `$CATALINA_BASE/conf/tomcat-users.xml`。）用户、密码以及相应角色通常可利用 JMX 进行动态编辑，更改结果会加以保存并立刻反映在 XML 文档中。

Realm 元素属性

跟之前讨论的一样，为了配置 UserDatabaseRealm，需要在 `$CATALINA_BASE/conf/server.xml` 中创建 `<Realm>` 元素。关于 UserDatabaseRealm 中的属性定义可参看 Realm 配置文档。

用户文件格式

用户文件使用的格式与 MemoryRealm所使用的相同。

范例

默认的 Tomcat 安装已经配置了内嵌在 `<Engine>` 元素中的 UserDatabaseRealm，因而可以将其应用于所有的虚拟主机和 Web 应用中。默认的 `conf/tomcat-users.xml` 文件内容为：

```
<tomcat-users>
  <user username="tomcat" password="tomcat" roles="tomcat" />
  <user username="role1"  password="tomcat" roles="role1"  />
  <user username="both"   password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

特别注意事项

使用 UserDatabaseRealm 需要遵循以下规则：

- 当 Tomcat 首次启动时，它会从用户文件中加载所有已定义的用户及其相关信息。假如对该用户文件中的数据进行修改，则只有重启 Tomcat 后才能生效。这些修改并不是通过 UserDatabase 数据源来完成的，是由 Tomcat 所提供的通过 JMX 访问的 MBean 来实现的。
- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。

MemoryRealm

简介

MemoryRealm 是一种对 Tomcat 的 Realm 接口的简单演示实现，并不是针对生产环境而设计的。在启动时，MemoryRealm 会从 XML 文档中加载所有的用户信息及其相关的角色信息（默认该文档位于 `$CATALINA_BASE/conf/tomcat-users.xml`）。只有重启 Tomcat 才能使对该文件作出的修改生效。

Realm 元素属性

跟之前讨论的一样，为了配置 MemoryRealm，需要在 `$CATALINA_BASE/conf/server.xml` 中创建 `<Realm>` 元素。关于 MemoryRealm 中的属性定义可参看 Realm 配置文档。

用户文件格式

用户文件包含下列属性。默认情况下，`conf/tomcat-users.xml` 必须是一个 XML 文件，并且带有一个根元素：`<tomcat-users>`。每一个有效用户都有一个内嵌在根元素中的 `<user>` 元素。

- **name** 用户登录所用的用户名。
- **password** 用户登录所用的密码。如果 `<Realm>` 元素中没有设置 `digest` 属性，则采用明文密码，否则就设置为摘要式密码，如之前讨论的那样。
- **roles** 以逗号分隔的用户角色名列表。

特别注意事项

使用 MemoryRealm 需要注意以下规则：

- 当 Tomcat 首次启动时，它会从用户文件中加载所有已定义的用户及其相关信息。假如对该用户文件中的数据进行修改，则只有重启 Tomcat 后才能生效。
- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。
- 应用负责管理**users**（用户表）和**user roles**（用户角色表）中的信息。Tomcat 没有提供任何内置功能来维护这两种表。

JAASRealm

简介

JAASRealm 是 Tomcat 的 `Realm` 接口的一种实现，通过 Java Authentication & Authorization Service（JAAS，Java身份验证与授权服务）架构来实现对用户身份的验证。JAAS 架构现已加入到标准的 Java SE API 中。

通过 JAASRealm，开发者实际上可以将任何安全的 Realm 与 Tomcat 的 CMA 一起组合使用。

JAASRealm 是 Tomcat 针对基于 JAAS 的 J2EE 1.4 的 J2EE 认证框架的原型实现，基于 JCP Specification Request 196，从而能够增强容器管理安全性，并且能促进“可插拔的”认证机制，该认证机制能够实现与容器的无关性。

根据 JAAS 登录模块和准则（参见 `javax.security.auth.spi.LoginModule` 与 `javax.security.Principal` 的相关说明），你可以自定义安全机制，或者将第三方的安全机制与 Tomcat 所实现的 CMA 相集成。

快速入门

为了利用自定义的 JAAS 登录模块使用 JAASRealm，需要执行如下步骤：

1. 编写自己的 JAAS 登录模块。在开发自定义登录模块时，将通过 JAAS 登录上下文对基于 JAAS [2]的 User 和 Role 类管理。注意，JAASRealm 内建的 `CallbackHandler` 目前只能识别 `NameCallback` 和 `PasswordCallback`。

   2. 详情请参看 JAAS 认证教程 与 JAAS 登录模块开发教程。

2. 尽管 JAAS 并未明确指定，但你也应该为用户和角色创建不同的类来加以区分，它们都应该扩展自 `javax.security.Principal`，从而使 Tomcat 明白从登录模块中返回的规则究竟是用户还是角色（参看 `org.apache.catalina.realm.JAASRealm` 相关描述）。不管怎样，第一个返回的规则总被认为是用户规则。

3. 将编译好的类指定在 Tomcat 的类路径中。

4. 为 Java 建立一个 login.config 文件（参见 JAAS LoginConfig 文件）。将其位置指定给 JVM，从而便于 Tomcat 明确它的位置。例如，设置如下环境变量：
   `JAVA OPTS=$JAVA OPTS -Djava.security.auth.login.config==$CATALINA_BASE/conf/jaas.config`
5. 为了保护一些资源，在 web.xml 中配置安全限制。
6. 在 server.xml 中配置 JAASRealm 模块。
7. 重启 Tomcat（如果它正在运行）。

Realm 元素属性

在上述步骤中，为了配置步骤 6 以上的 JAASRealm，需要创建一个 `<Realm>` 元素，并将其内嵌在 `<Engine>` 元素中的 `$CATALINA_BASE/conf/server.xml` 文件内。关于 JAASRealm 中的属性定义可参看 Realm 配置文档。

范例

下例是 server.xml 中的一截代码段：

```
<Realm className="org.apache.catalina.realm.JAASRealm"
                appName="MyFooRealm"
    userClassNames="org.foobar.realm.FooUser"
     roleClassNames="org.foobar.realm.FooRole"/>
```

完全由登录模块负责创建并保存用于表示用户规则的 User 与 Role 对象（`javax.security.auth.Subject`）。如果登录模块不仅无法创建用户对象，而且也无法抛出登录异常，Tomcat CMA 就会失去作用，所在页面就会变成 `http://localhost:8080/myapp/j_security_check` 或其他未指明的页面。

JAAS 方法具有双重的灵活性：

- 你可以在自定义的登录模块后台执行任何所需的进程。
- 通过改变配置以及重启服务器，你可以插入一个完全不同的登录模块，不需要对应用做出任何改动。

特别注意事项

- 当用户首次访问一个受保护资源时，Tomcat 会调用这一 `Realm` 的 `authenticate()` 方法。
- 一旦用户认证成功，在登录后，该用户（及其相应角色）就将缓存在 Tomcat 中。（对于以表单形式的认证，这意味着直到会话超时或者无效才会过期；对于基本形式的验证，意味着直到用户关闭浏览器才会过期。）在会话序列化期间不会保存或重置缓存的用户。对已认证用户的数据库信息进行的任何改动都不会生效，直到该用户下次登录。
- 和其他 Realm 实现一样，如果 `server.xml` 中的 `<Realm>` 元素包含一个 `digest` 属性，则支持摘要式密码。JAASRealm 的 `CallbackHandler` 将先于将密码传回 `LoginModule` 之前，对密码进行摘要式处理。

CombinedRealm

简介

CombinedRealm 是一种 Tomcat 的 Realm 实现，通过一个或多个子 Realm 进行用户验证。

通过 CombinedRealm，开发者能够将多个 Realm（同一或不同类型） 组合起来使用，从而用于验证多种数据源，而且万一当其中一个 Realm 失败，或其他一些操作需要多个 Realm 时，它还能提供回滚处理。

子 Realm 是通过在定义 CombineRealm 的 `Realm` 元素中内嵌 `Realm` 元素来实现的。验证操作会按照 Realm 元素的叠加顺序来逐个进行。对逐个 Realm 进行验证，从而就能充分证明用户的身份。

Realm 元素属性

为了配置 CombinedRealm，需要创建一个 `<Realm>` 元素，并将其内嵌在 `<Engine>` 或 `<Host>` 元素中的 `$CATALINA_BASE/conf/server.xml` 文件内。同样，你也可以将其内嵌到 `context.xml` 文件下的 `<Context>` 节点。

范例

下面是 server.xml 中的一段代码，综合使用了 UserDatabaseRealm 和 DataSourceRealm：

```
<Realm className="org.apache.catalina.realm.CombinedRealm" >
   <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase"/>
   <Realm className="org.apache.catalina.realm.DataSourceRealm"
             dataSourceName="jdbc/authority"
             userTable="users" userNameCol="user_name" userCredCol="user_pass"
             userRoleTable="user_roles" roleNameCol="role_name"/>
</Realm>
```

LockOutRealm

简介

LockOutRealm 是一个 Tomcat 的 Realm 实现，它扩展了 CombinedRealm，假如在某一段时间内出现很多验证失败，则它能够提供锁定用户的功能。

为了确保操作的正确性，该 Realm 允许出现较合理的同步。

该 Realm 并不需要对底层的 Realm 或与其相关的用户存储机制进行任何改动。它会记录失败的登录，包括那些因为用户不存在的登录。为了防止无效用户通过精心设计的请求而实施的 DOS 攻击（从而造成缓存增加），没有通过验证的用户所在列表的容量受到了严格的限制。

子 Realm 是通过在定义 LockOutRealm 的 `Realm` 元素中内嵌 `Realm` 元素来实现的。验证操作会按照 Realm 元素的叠加顺序来逐个进行。对逐个 Realm 进行验证，从而就能充分证明用户的身份。

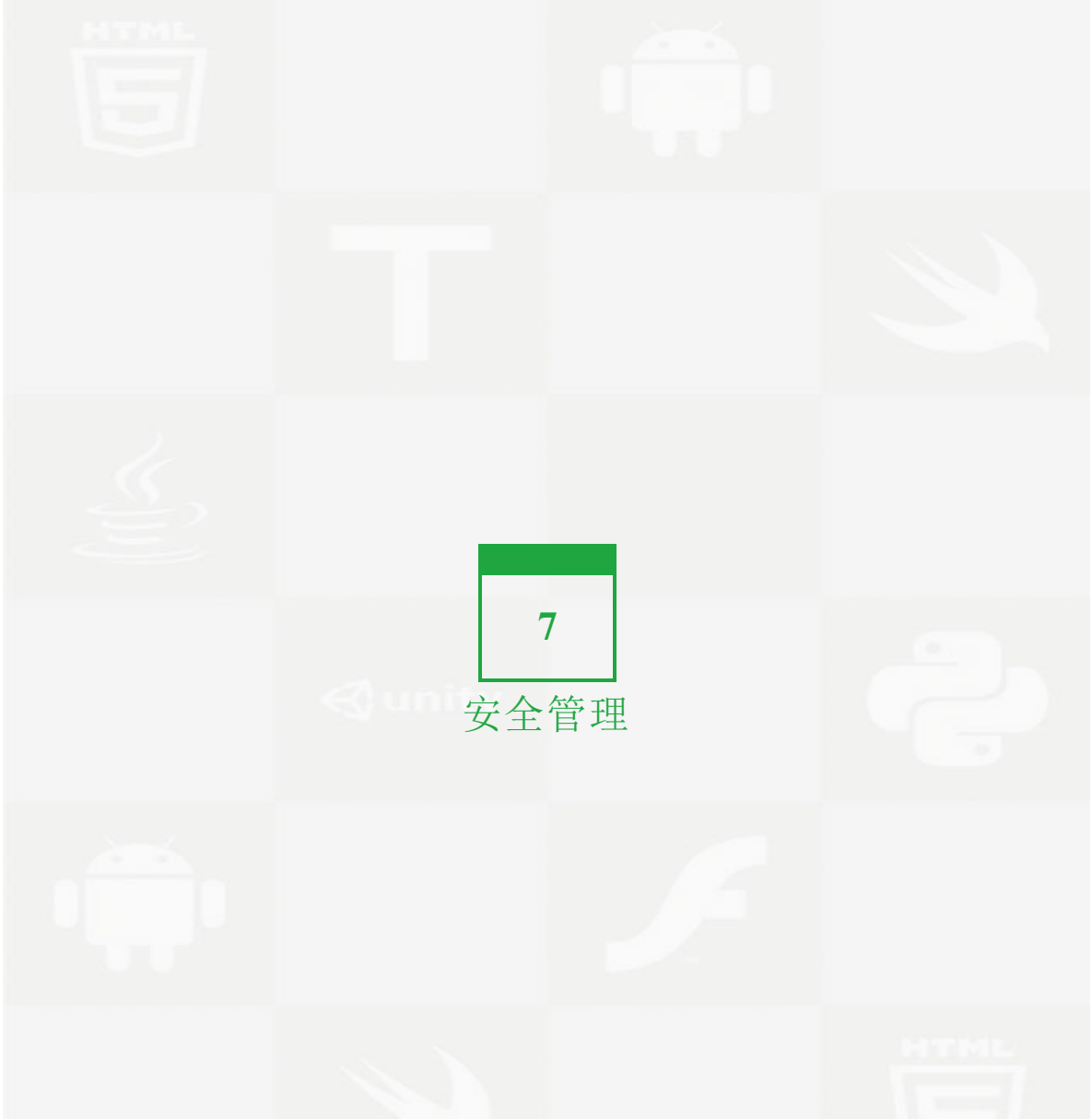
Realm 元素属性

为了配置 CombinedRealm，需要创建一个 `<Realm>` 元素，并将其内嵌在 `<Engine>` 或 `<Host>` 元素中的 `$CATALINA_BASE/conf/server.xml` 文件内。同样，你也可以将其内嵌到 `context.xml` 文件下的 `<Context>` 节点。关于 LockOutRealm 中的属性定义可参看 Realm 配置文档。

范例

下面是 server.xml 中的一段代码，为 UserDatabaseRealm 添加了锁定功能。

```
<Realm className="org.apache.catalina.realm.LockOutRealm" >
   <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase"/>
</Realm>
```



7

安全管理

背景知识

Java 的 **SecurityManager** 能让 Web 浏览器在它自身的沙盒中运行小型应用（applet），从而具有防止不可信代码访问本地文件系统的文件以及防止其连接到主机，而不是加载该应用的位置，等等。如同 SecurityManager 能防止不可信的小型应用在你的浏览器上运行，运行 Tomcat 时，使用 SecurityManager 也能保护服务器，使其免受木马型的 applet、JSP、JSP Bean 以及标签库的侵害，甚至也可以防止由于无意中的疏忽所造成的问题。

假设网站有一位经授权可发布 JSP 的用户，他在无意中将下面这些代码加入了 JSP 中：

```
<% System.exit(1); %>
```

每当 Tomcat 执行这个 JSP 文件时，Tomcat 都会退出。Java 的 SecurityManager 构成了系统管理员保证服务器安全可靠的另一道防线。

警告：使用 Tomcat 代码库时会执行一个安全审核。大多数关键包已受到保护，新的安全包保护机制已经实施。然而，在允许不可信用户发布 Web 应用、JSP、servlet、bean 或标签库之前，你仍要反复确定自己配置的 SecurityManager 是否满足了要求。但不管怎么说，利用 **SecurityManager** 来运行 **Tomcat** 肯定比没有它好得多

权限

权限类用于定义 Tomcat 加载的类所具有的权限。标准 JDK 中包含了很多标准权限类，你还可以针对自己的 Web 应用自定义权限类。Tomcat 支持这两种技术。

标准权限

关于适用于 Tomcat 的标准系统 SecurityManager 权限类，以下仅是一个简短的总结。详情请查看http://docs.oracle.com/javase/7/docs/technotes/guides/security/。

- **java.util.PropertyPermission**——控制对 JVM 属性的读/写，比如说 `java.home`。
- **java.lang.RuntimePermission**——控制一些系统/运行时函数的使用，比如 `exit()` 和 `exec()`。另外也控制包的访问/定义。
- **java.io.FilePermission**——控制对文件和目录的读/写/执行。
- **java.net.SocketPermission**——控制网络套接字的使用。
- **java.net.NetPermission**——控制组播网络连接的使用。
- **java.lang.reflect.ReflectPermission**——控制类反射的使用。
- **java.security.SecurityPermission**——控制对 Security 方法的访问。
- **java.security.AllPermission**——允许访问任何权限，仿佛没有 SecurityManager。

Tomcat 自定义权限

Tomcat 使用了一个自定义权限类 **org.apache.naming.JndiPermission**。该权限能够控制对 JNDI 命名的基于文件的资源的可读访问。权限名就是 JNDI 名，无任何行为。后面的 `*` 可以用来在授权时进行模糊匹配。比如，可以在策略文件中加入以下内容：

```
permission  org.apache.naming.JndiPermission  "jndi://localhost/examples/*";
```

从而为每个部署的 Web 应用动态生成这样的权限项，允许它们读取自己的静态资源，而不允许读取其他的文件（除非显式地赋予这些文件权限）。

另外，Tomcat 还能动态生成下面这样的文件权限。

```
permission java.io.FilePermission "** your application context**", "read";

permission java.io.FilePermission
  "** application working directory**", "read,write";
permission java.io.FilePermission
  "** application working directory**/-", "read,write,delete";
```

***application working directory** 是部署应用所用的文件夹或 WAR 文件。 **application working directory** 是应 Servlet 规范需要而提供给应用的暂时性目录。

利用 SecurityManager 配置 Tomcat

策略文件格式

Java SecurityManager 所实现的安全策略配置在 `$CATALINA_BASE/conf/catalina.policy` 文件中。该文件完全替代了 JDK 系统目录中提供的 `java.policy` 文件。既可以手动编辑 `catalina.policy` 文件，也可以使用Java 1.2 或以后版本附带的 policytool 应用。

`catalina.policy` 文件中的项使用标准的 `java.policy` 文件格式，如下所示：

```
// 策略文件项范例

grant [signedBy <signer>,] [codeBase <code source>] {
  permission  <class>  [<name> [, <action list>]];
};
```

signedBy 和 **codeBase** 两项在授予权限时是可选项。注释行以 `//` 开始，在当前行结束。`codeBase` 以 URL 的形式。对于文件 URL，可以使用 `${java.home}` 与 `${catalina.home}` 属性（这些属性代表的是使用 `JAVA_HOME`、`CATALINA_HOME` 和 `CATALINA_BASE` 环境变量为这些属性定义的目录路径）。

默认策略文件

默认的 `$CATALINA_BASE/conf/catalina.policy` 文件如下所示：

```
// Licensed to the Apache Software Foundation (ASF) under one or more
// contributor license agreements.  See the NOTICE file distributed with
// this work for additional information regarding copyright ownership.
// The ASF licenses this file to You under the Apache License, Version 2.0
// (the "License"); you may not use this file except in compliance with
// the License.  You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ============================================================================
// catalina.policy - Security Policy Permissions for Tomcat
//
// This file contains a default set of security policies to be enforced (by the
// JVM) when Catalina is executed with the "-security" option.  In addition
// to the permissions granted here, the following additional permissions are
// granted to each web application:
//
// * Read access to the web application's document root directory
// * Read, write and delete access to the web application's working directory
// ============================================================================


// ========== SYSTEM CODE PERMISSIONS =========================================


// These permissions apply to javac
```

```
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};

// These permissions apply to all shared system extensions
grant codeBase "file:${java.home}/jre/lib/ext/-" {
        permission java.security.AllPermission;
};

// These permissions apply to javac when ${java.home] points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/../lib/-" {
        permission java.security.AllPermission;
};

// These permissions apply to all shared system extensions when
// ${java.home} points at $JAVA_HOME/jre
grant codeBase "file:${java.home}/lib/ext/-" {
        permission java.security.AllPermission;
};


// ========== CATALINA CODE PERMISSIONS =======================================


// These permissions apply to the daemon code
grant codeBase "file:${catalina.home}/bin/commons-daemon.jar" {
        permission java.security.AllPermission;
};

// These permissions apply to the logging API
// Note: If tomcat-juli.jar is in ${catalina.base} and not in ${catalina.home},
// update this section accordingly.
//  grant codeBase "file:${catalina.base}/bin/tomcat-juli.jar" {..}
grant codeBase "file:${catalina.home}/bin/tomcat-juli.jar" {
        permission java.io.FilePermission
         "${java.home}${file.separator}lib${file.separator}logging.properties",
"read";

        permission java.io.FilePermission

"${catalina.base}${file.separator}conf${file.separator}logging.properties", "read";
        permission java.io.FilePermission
         "${catalina.base}${file.separator}logs", "read, write";
        permission java.io.FilePermission
         "${catalina.base}${file.separator}logs${file.separator}*", "read, write";

        permission java.lang.RuntimePermission "shutdownHooks";
        permission java.lang.RuntimePermission "getClassLoader";
        permission java.lang.RuntimePermission "setContextClassLoader";

        permission java.lang.management.ManagementPermission "monitor";

        permission java.util.logging.LoggingPermission "control";

        permission java.util.PropertyPermission "java.util.logging.config.class",
"read";
        permission java.util.PropertyPermission "java.util.logging.config.file",
"read";
        permission java.util.PropertyPermission
"org.apache.juli.AsyncLoggerPollInterval", "read";
        permission java.util.PropertyPermission
"org.apache.juli.AsyncMaxRecordCount", "read";
```

```
        permission java.util.PropertyPermission
"org.apache.juli.AsyncOverflowDropType", "read";
        permission java.util.PropertyPermission
"org.apache.juli.ClassLoaderLogManager.debug", "read";
        permission java.util.PropertyPermission "catalina.base", "read";

        // Note: To enable per context logging configuration, permit read access to
        // the appropriate file. Be sure that the logging configuration is
        // secure before enabling such access.
        // E.g. for the examples web application (uncomment and unwrap
        // the following to be on a single line):
        // permission java.io.FilePermission "${catalina.base}${file.separator}
        //  webapps${file.separator}examples${file.separator}WEB-INF
        //  ${file.separator}classes${file.separator}logging.properties", "read";
};

// These permissions apply to the server startup code
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
        permission java.security.AllPermission;
};

// These permissions apply to the servlet API classes
// and those that are shared across all class loaders
// located in the "lib" directory
grant codeBase "file:${catalina.home}/lib/-" {
        permission java.security.AllPermission;
};


// If using a per instance lib directory, i.e. ${catalina.base}/lib,
// then the following permission will need to be uncommented
// grant codeBase "file:${catalina.base}/lib/-" {
//         permission java.security.AllPermission;
// };


// ========== WEB APPLICATION PERMISSIONS =====================================


// These permissions are granted by default to all web applications
// In addition, a web application will be given a read FilePermission
// for all files and directories in its document root.
grant {
    // Required for JNDI lookup of named JDBC DataSource's and
    // javamail named MimePart DataSource used to send mail
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "java.naming.*", "read";
    permission java.util.PropertyPermission "javax.sql.*", "read";

    // OS Specific properties to allow read access
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";

    // JVM properties to allow read access
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
```

```
    permission java.util.PropertyPermission "java.specification.version", "read";
    permission java.util.PropertyPermission "java.specification.vendor", "read";
    permission java.util.PropertyPermission "java.specification.name", "read";

    permission java.util.PropertyPermission "java.vm.specification.version",
"read";
    permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
    permission java.util.PropertyPermission "java.vm.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.version", "read";
    permission java.util.PropertyPermission "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";

    // Required for OpenJMX
    permission java.lang.RuntimePermission "getAttribute";

    // Allow read of JAXP compliant XML parser debug
    permission java.util.PropertyPermission "jaxp.debug", "read";

    // All JSPs need to be able to read this package
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.tomcat";

    // Precompiled JSPs need access to these packages.
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.jasper.el";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.jasper.runtime";
    permission java.lang.RuntimePermission
     "accessClassInPackage.org.apache.jasper.runtime.*";

    // Precompiled JSPs need access to these system properties.
    permission java.util.PropertyPermission
     "org.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER", "read";
    permission java.util.PropertyPermission
     "org.apache.el.parser.COERCE_TO_ZERO", "read";

    // The cookie code needs these.
    permission java.util.PropertyPermission
     "org.apache.catalina.STRICT_SERVLET_COMPLIANCE", "read";
    permission java.util.PropertyPermission
     "org.apache.tomcat.util.http.ServerCookie.STRICT_NAMING", "read";
    permission java.util.PropertyPermission
     "org.apache.tomcat.util.http.ServerCookie.FWD_SLASH_IS_SEPARATOR", "read";

    // Applications using Comet need to be able to access this package
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.comet";

    // Applications using WebSocket need to be able to access these packages
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.tomcat.websocket";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.tomcat.websocket.server";
};


// The Manager application needs access to the following packages to support the
// session display functionality. These settings support the following
// configurations:
// - default CATALINA_HOME == CATALINA_BASE
// - CATALINA_HOME != CATALINA_BASE, per instance Manager in CATALINA_BASE
// - CATALINA_HOME != CATALINA_BASE, shared Manager in CATALINA_HOME
```

```
grant codeBase "file:${catalina.base}/webapps/manager/-" {
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.ha.session";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.manager";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.manager.util";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.util";
};
grant codeBase "file:${catalina.home}/webapps/manager/-" {
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.ha.session";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.manager";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.manager.util";
    permission java.lang.RuntimePermission
"accessClassInPackage.org.apache.catalina.util";
};

// You can assign additional permissions to particular web applications by
// adding additional "grant" entries here, based on the code base for that
// application, /WEB-INF/classes/, or /WEB-INF/lib/ jar files.
//
// Different permissions can be granted to JSP pages, classes loaded from
// the /WEB-INF/classes/ directory, all jar files in the /WEB-INF/lib/
// directory, or even to individual jar files in the /WEB-INF/lib/ directory.
//
// For instance, assume that the standard "examples" application
// included a JDBC driver that needed to establish a network connection to the
// corresponding database and used the scrape taglib to get the weather from
// the NOAA web server.  You might create a "grant" entries like this:
//
// The permissions granted to the context root directory apply to JSP pages.
// grant codeBase "file:${catalina.base}/webapps/examples/-" {
//      permission java.net.SocketPermission "dbhost.mycompany.com:5432",
"connect";
//      permission java.net.SocketPermission "*.noaa.gov:80", "connect";
// };
//
// The permissions granted to the context WEB-INF/classes directory
// grant codeBase "file:${catalina.base}/webapps/examples/WEB-INF/classes/-" {
// };
//
// The permission granted to your JDBC driver
// grant codeBase "jar:file:${catalina.base}/webapps/examples/WEB-
INF/lib/driver.jar!/-" {
//      permission java.net.SocketPermission "dbhost.mycompany.com:5432",
"connect";
// };
// The permission granted to the scrape taglib
// grant codeBase "jar:file:${catalina.base}/webapps/examples/WEB-
INF/lib/scrape.jar!/-" {
//      permission java.net.SocketPermission "*.noaa.gov:80", "connect";
// };
```

使用 SecurityManager 启动 Tomcat

一旦配置好了用于 SecurityManager 的 `catalina.policy` 文件，就可以使用 `-security` 选项启动带有 SecurityManager 的 Tomcat。

```
$CATALINA_HOME/bin/catalina.sh start -security     (Unix)
%CATALINA_HOME%\bin\catalina start -security       (Windows)
```

配置 Tomcat 中的包保护

从 Tomcat 5 开始，可以通过配置保护 Tomcat 内部包，使其免于被定义与访问。详情查看 http://www.oracle.com/technetwork/java/seccodeguide-139067.html。

警告：假如去除默认的包保护，可能会造成安全漏洞。

默认属性文件

默认的 `$CATALINA_BASE/conf/catalina.properties` 文件如下所示：

```
 #
 # List of comma-separated packages that start with or equal this string
 # will cause a security exception to be thrown when
 # passed to checkPackageAccess unless the
 # corresponding RuntimePermission ("accessClassInPackage."+package) has
 # been granted.
package.access=sun.,org.apache.catalina.,org.apache.coyote.,org.apache.tomcat.,
org.apache.jasper.
 #
 # List of comma-separated packages that start with or equal this string
 # will cause a security exception to be thrown when
 # passed to checkPackageDefinition unless the
 # corresponding RuntimePermission ("defineClassInPackage."+package) has
 # been granted.
 #
 # by default, no packages are restricted for definition, and none of
 # the class loaders supplied with the JDK call checkPackageDefinition.
 #
package.definition=sun.,java.,org.apache.catalina.,org.apache.coyote.,
org.apache.tomcat.,org.apache.jasper.
```

一旦为 SecurityManager 配置了 `catalina.properties` 文件，记得重启 Tomcat。

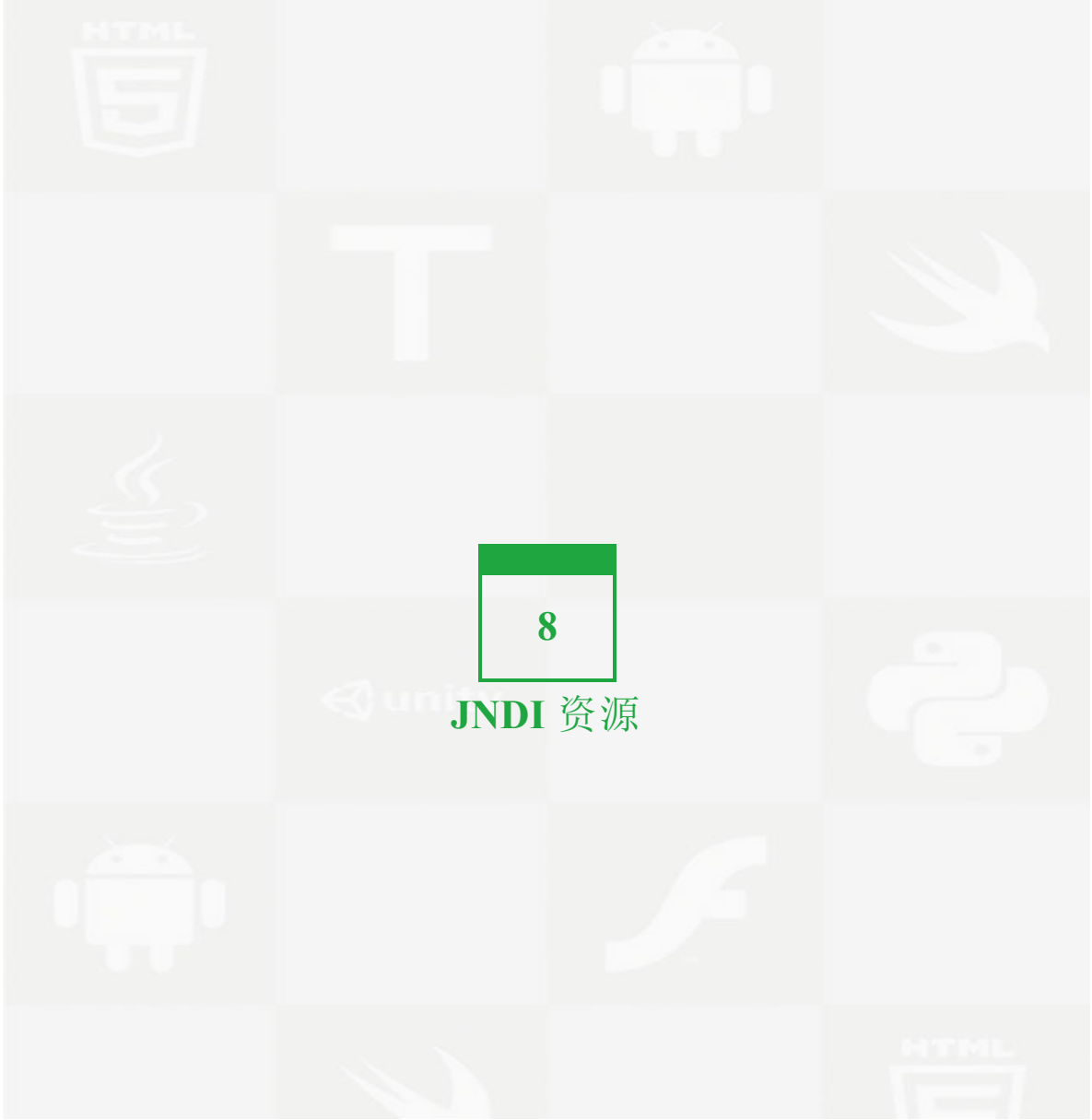
疑难解答

假如应用执行一个由于缺乏所需权限而被禁止的操作，当 SecurityManager 侦测到这种违规时，就会抛出 `AccessControLException` 或 `SecurityException` 异常。虽然调试缺失的权限是很有难度的，但还是有一个办法，那就是将在执行中制定的所有安全决策的调试输出打开，这需要在启动 Tomcat 之前设置一个系统属性。最简单的方法就是通过 `CATALINA_OPTS` 环境变量来实现，命令如下所示：

```
export CATALINA_OPTS=-Djava.security.debug=all     (Unix)
set CATALINA_OPTS=-Djava.security.debug=all        (Windows)
```

记住，一定要在启动 Tomcat 之前去做。

警告：这将生成很多兆的输出内容！但是，它能通过搜索关键字 `FAILED` 来锁定问题所在位置，确定需要检查的权限。此外，查阅 Java 安全文档可了解更多的可设置选项。



8

JNDI 资源

本章概述

Tomcat 为每个在其上运行的 Web 应用都提供了一个 JNDI 的 `InitialContext` 实现实例，它与Java 企业版应用服务器所提供的对应类完全兼容。Java EE 标准在 `/WEB-INF/web.xml` 文件中提供了一系列标准元素，用来引用或定义资源。

可通过下列规范了解如何编写针对 JNDI 的 API 以及 Java 企业版（Java EE）服务器所支持的功能，这也是 Tomcat 针对其所提供的服务而仿效的功能。

- Java 命名与目录接口（包括在 JDK 1.4 或更前的版本）
- Java EE 平台规范，查看其中的第5章：*Naming*（命名）

web.xml 配置

可在 Web 应用的部署描述符文件（`/WEB-INF/web.xml`）中使用下列元素来定义资源：

- `<env-entry>` 应用的环境项。一个可用于配置应用运行方式的单值参数。
- `<resource-ref>` 资源引用，通常是引用保存某种资源的对象工厂，比如 JDBC `DataSource` 或 JavaMail `Session` 这样的资源；或者引用配置在 Tomcat 中的自定义对象工厂中的资源。
- `<resource-env-ref>` 资源环境引用。Servlet 2.4 所添加的一种新 `resource-ref`，它简化了不需要认证消息的资源的配置。

有了这些，Tomcat 就能利用适宜的资源工厂来创建资源，再也不需要其他配置信息了。Tomcat 将使用 `/WEB-INF/web.xml` 中的信息来创建资源。

另外，Tomcat 还提供了一些用于 JNDI 的特殊选项，它们没有指定在 web.xml 中。比如，其中包括的 `closeMethod` 能在 Web 应用停止时，迅速清除 JNDI 资源；`singleton` 控制是否会在每次 JNDI 查找时创建资源的新实例。要想使用这些配置选项，资源必须指定在 Web 应用的 `<Context>` 元素内，或者位于 `$CATALINA_BASE/conf/server.xml` 的 `<GlobalNamingResources>` 元素中。

context.xml 配置

如果 Tomcat 无法确定合适的资源工厂，并且/或者需要额外的配置信息，就必须在 Tomcat 创建资源之前指定好额外的具体配置。Tomcat 特定资源配置应位于 `<Context>` 元素内，它可以指定在 `$CATALINA_BASE/conf/server.xml`，或者，最好放在每个 Web 应用的上下文 XML 文件中（`META-INF/context.xml`）。

要想完成 Tomcat 的特定资源配置，需要使用 `<Context>` 元素中的下列元素：

- `<Environment>` 对将通过 JNDI 的 `InitialContext` 方法暴露给 Web 应用的环境项的名称与数值加以配置（等同于 Web 应用部署描述符文件中包含了一个 `<env-entry>` 元素）。
- `<Resource>` 定义应用所能用到的资源名称和数据类型（等同于 Web 应用部署描述符文件中包含了一个 `<resource-ref>` 元素）。
- `<ResourceLink>` 添加一个链接，使其指向全局 JNDI 上下文中定义的资源。使用资源链接可以使 Web 应用访问定义在 `<Server>` 元素中子元素 `<GlobalNamingResources>` 中的资源。
- `<Transaction>` 添加一个资源工厂，用于对从 `java:comp/UserTransaction` 获得的 UserTransaction 接口进行实例化。

以上这些元素内嵌于 `<Context>` 元素中，而且是与特定应用相关联的。

如果资源已经定义在 `<Context>` 元素中，那就不必再在部署描述符文件中定义它了。但建议在部署描述符文件中保留相关项，以便记录应用资源需求。

加入同样一个资源名称既被定义在 Web 应用部署描述符文件的 `<env-entry>` 元素中，又被定义在 Web 应用的 `<Context>` 元素的 `<Environment>` 元素内，那么只有当相应的 `<Environment>` 元素允许时（将其中的 `override` 属性设为 `true`），部署描述符文件中的值才会优先对待。

全局配置

Tomcat 为整个服务器维护着一个全局资源的独立命名空间。这些全局资源配置在 `$CATALINA BASE/conf/server.xml` 的 `<GlobalNamingResources>` 元素内。可以使用 `<ResourceLink>` 将这些资源暴露给 Web 应用，以便在每一应用上下文中将其包含进来。

如果资源已经定义在 `<Context>` 元素中，那就不必再在部署描述符文件中定义它了。但建议在部署描述符文件中保留相关项，以便记录应用资源需求。

使用资源

当 Web 应用最初部署时，就配置 `InitialContext`，使其可被 Web 应用的各组件所使用（只读访问）。JNDI 命名空间的 `java:comp/env` 部分中包含着所有的配置项与资源，所以访问资源（在下例中，就是一个 JDBC 数据源）应按如下形式进行：

```
// 获取环境命名上下文
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");

// 查找数据源
DataSource ds = (DataSource)
  envCtx.lookup("jdbc/EmployeeDB");

// 分配并使用池中的连接
Connection conn = ds.getConnection();
... use this connection to access the database ...
conn.close();
```

Tomcat 标准资源工厂

Tomcat 包含一系列资源工厂，能为 Web 应用提供各种服务，而且无需修改 Web 应用或部署描述符文件即能灵活配置（通过 `<Context>` 元素）。下面所列出的每一小节都详细介绍了标准资源工厂的配置与用途。

要想了解如何创建、安装、配置和使用你自己的自定义资源工厂类，请参看添加自定义资源工厂。

注意：在标准资源工厂中，只有“JDBC DataSource”和“User Transaction”工厂可适用于其他平台，而且这些平台必须实现了 Java EE 规范。而其他所有标准资源工厂，以及你自己编写的自定义资源工厂，则都是 Tomcat 所专属的，不适用于其他容器。

一般 JavaBean 资源

简介

该资源工厂能创建出任何符合标准 JavaBean 命名规范[1]的 Java 类的对象。如果工厂的 `singleton` 属性被设为 `false`，那么每当对该项进行 `lookup` 时，资源工厂将会创建出适合的 bean 类的新实例。

1. 标准的 JavaBean 命名规范，比如：构造函数没有任何参数，属性设置器遵守 `setFoo()` 命名模式，等等。

使用该功能所需的步骤将在下文介绍。

创建 JavaBean 类

创建一个 JavaBean 类，在每次查找资源工厂时，就创建它的实例。比如，假设你创建了一个名叫 `com.mycompany.MyBean` 的 JavaBean 类，如下所示：

```
package com.mycompany;

public class MyBean {

  private String foo = "Default Foo";

  public String getFoo() {
    return (this.foo);
  }

  public void setFoo(String foo) {
    this.foo = foo;
  }

  private int bar = 0;

  public int getBar() {
    return (this.bar);
  }

  public void setBar(int bar) {
    this.bar = bar;
  }


}
```

声明资源需求

接下来，修改 Web 应用部署描述符文件（`/WEB-INF/web.xml`），声明 JNDI 名称，并据此请求该 Bean 类的新实例。最简单的方法是使用 `<resource-env-ref>` 元素，如下所示：

```
<resource-env-ref>
  <description>
    Object factory for MyBean instances.
  </description>
  <resource-env-ref-name>
    bean/MyBeanFactory
  </resource-env-ref-name>
  <resource-env-ref-type>
    com.mycompany.MyBean
  </resource-env-ref-type>
</resource-env-ref>
```

警告：一定要遵从 Web 应用部署描述符文件中 DTD 所需要的元素顺序。关于这点，可参看Servlet 规范中的解释。

使用资源

资源引用的典型用例如下所示：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");

writer.println("foo = " + bean.getFoo() + ", bar = " +
               bean.getBar());
```

配置 **Tomcat** 资源工厂

为了配置 Tomcat 资源工厂，为 Web 应用的 `<Context>` 元素添加下列元素：

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean"
            factory="org.apache.naming.factory.BeanFactory"
            bar="23"/>
  ...
</Context>
```

注意这里的资源名称，这里 `bean/MyBeanFactory` 必须跟部署描述符文件中所指定的值完全一样。这里还初始化了 `bar` 属性值，从而当返回新的 bean 时，`setBar(23)` 就会被调用。由于我们没有初始化 `foo` 属性（虽然我们完全可以这么做），所以 bean 依然采用构造函数中设置的默认值。

假设我们的 Bean 如下所示：

```
package com.mycompany;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class MyBean2 {

  private InetAddress local = null;

  public InetAddress getLocal() {
    return local;
  }
```

```
  public void setLocal(InetAddress ip) {
    local = ip;
  }

  public void setLocal(String localHost) {
    try {
      local = InetAddress.getByName(localHost);
    } catch (UnknownHostException ex) {
    }
  }

  private InetAddress remote = null;

  public InetAddress getRemote() {
    return remote;
  }

  public void setRemote(InetAddress ip) {
    remote = ip;
  }

  public void host(String remoteHost) {
    try {
      remote = InetAddress.getByName(remoteHost);
    } catch (UnknownHostException ex) {
    }
  }

}
```

该 Bean 有两个 `InetAddress` 类型的属性。第一个属性 `local` 还有第二种 setter 方法，传入的是一个字符串参数。默认 Tomcat BeanFactory 会使用自动侦测到的 setter 方法，并将其参数类型作为属性类型，然后抛出一个 NamingException（命名异常），因为它还没有准备好将给定字符串值转化为 `InetAddress`。我们可以让 Tomcat BeanFactory 使用其他的 setter 方法，如下所示：

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean2"
            factory="org.apache.naming.factory.BeanFactory"
            forceString="local"
            local="localhost"/>
  ...
</Context>
```

bean 属性 `remote` 也可以从字符串中设置，但必须使用非标准方法 `host`。如下设置 `local` 和 `remote`：

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean2"
            factory="org.apache.naming.factory.BeanFactory"
            forceString="local,remote=host"
            local="localhost"
            remote="tomcat.apache.org"/>
  ...
</Context>
```

如上所示，可以利用逗号作分隔符，将多个属性描述串联在一起放在 `forceString` 中。每一属性描述要么只包含属性名，要么由 `name = method` 的结构所组成。对于前者的情况，BeanFactory 会直接调用属性名的 setter 方法；而对于后者，则通过调用方法 `method` 来设置属性名 `name`。对于 `String` 或基本类型，或者相

应的基本包装器类的属性，不必使用 `forceString`。会自动侦测正确的 setter 并实施参数类型转换。

UserDatabase 资源

简介

UserDatabase 资源通常被配置成通过 UserDataBase Realm 所使用的全局资源。Tomcat 包含一个 UserDatabaseFactoory，能够创建基于 XML 文件（通常是 `tomcat-users.xml`）的 UserDatabase 资源。

建立全局的 UserDataBase 资源的步骤如下。

创建/编辑 **XML**文件

XML 文件通常位于 `$CATALINA BASE/conf/tomcat-users.xml`，但也可以放在文件系统中的任何位置。我们建议把该文件放在 `$CATALINA_BASE/conf`。典型的 XML 应如下所示：

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
</tomcat-users>
```

声明资源

接下来，修改 `$CATALINA_BASE/conf/server.xml` 来创建基于此文件的 UserDataBase 资源。如下所示：

```
<Resource name="UserDatabase"
          auth="Container"
          type="org.apache.catalina.UserDatabase"
          description="User database that can be updated and saved"
          factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
          pathname="conf/tomcat-users.xml"
          readonly="false" />
```

属性 `pathname` 可以采用绝对路径或相对路径。相对路径意味着是相对于 `$CATALINA_BASE`。

`readonly` 属性是可选属性，如果不采用，则默认为 `true`。如果该 XML 文件可写，那么当 Tomcat 开启时，就会被修改。警告：当该文件被修改后，它会继承 Tomcat 目前运行用户的默认文件权限。所以要确保这样做是否能保持应用的安全性。

配置 **Realm**

配置 UserDatabase Realm 以便使用该资源，详情可参看 Realm 配置文档

JavaMail 会话

简介

很多 Web 应用都会把发送电子邮件作为系统的必备功能。JavaMail API 可以让这一过程变得相对简单些，但需要很多的配置细节，客户端应用必须知道的（包括用于发送消息的 SMTP 主机的名称）。

Tomcat 所包含的标准资源工厂可以为你创建 `javax.mail.Session` 会话实例，并且已经配置好连接到 SMTP 服务器上，从而使应用完全与电子邮件配置环境相隔离，不受后者变更的影响，无论何时，只需请求并接受预配置的会话即可。

所需步骤如下所示。

声明资源需求

首先应该做的是修改 Web 应用的部署描述符文件（`/WEB-INF/web.xml`），声明 JNDI 名称以便借此查找预配置会话。按照惯例，所有这样的名字都应该解析到 `mail` 子上下文（相对于标准的 `java:comp/env` 命名上下文而言的，这个命名上下文是所有资源工厂的基准。）典型的 `web.xml` 项应该如下所示：

```
<resource-ref>
  <description>
    Resource reference to a factory for javax.mail.Session
    instances that may be used for sending electronic mail
    messages, preconfigured to connect to the appropriate
    SMTP server.
  </description>
  <res-ref-name>
    mail/Session
  </res-ref-name>
  <res-type>
    javax.mail.Session
  </res-type>
  <res-auth>
    Container
  </res-auth>
</resource-ref>
```

警告：一定要遵从 Web 应用部署描述符文件中 DTD 所需要的元素顺序。关于这点，可参看Servlet 规范中的解释。

使用资源

资源引用的典型用例如下所示：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
Session session = (Session) envCtx.lookup("mail/Session");

Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(request.getParameter("from")));
InternetAddress to[] = new InternetAddress[1];
to[0] = new InternetAddress(request.getParameter("to"));
message.setRecipients(Message.RecipientType.TO, to);
message.setSubject(request.getParameter("subject"));
message.setContent(request.getParameter("content"), "text/plain");
Transport.send(message);
```

注意，该应用所用的资源引用名与 Web 应用部署符中声明的完全相同。这是与下文会讲到的 `<Context>` 元素里所配置的资源工厂相匹配的。

配置 **Tomcat** 资源工厂

为了配置 Tomcat 的资源工厂，在 `<Context>` 元素中添加以下元素：

```
<Context ...>
```

```
  ...
  <Resource name="mail/Session" auth="Container"
            type="javax.mail.Session"
            mail.smtp.host="localhost"/>
  ...
</Context>
```

注意，资源名（在这里，是 `mail/Session`）必须与 Web 应用部署描述符文件中所指定的值相匹配。对于 `mail.smtp.host` 参数值，可以用为你的网络提供 SMTP 服务的服务器来自定义。

额外的资源属性与值将转换成相关的属性及值，并被传入 `javax.mail.Session.getInstance(java.util.Properties)`，作为参数集 `java.util.Properties` 中的一部分。除了 JavaMail 规范附件A中所定义的属性之外，个别的提供者可能还支持额外的属性。

如果资源配置中包含 `password` 属性，以及 `mail.smtp.user` 或 `mail.user` 属性，那么 Tomcat 资源工厂将配置并添加 `javax.mail.Authenticator` 到邮件会话中。

安装 **JavaMail** 库

下载 JavaMail API

解压缩文件分发包，将 mail.jar 放到 $CATALINA_HOME/lib 中，从而使 Tomcat 能在邮件会话资源初始化期间能够使用它。注意：不能将这一文件同时放在 $CATALINA_HOME/lib 和 Web 应用的 /lib 文件夹中，否则就会出错，只能将其放在 $CATALINA_HOME/lib 中。

重启 **Tomcat**

为了能让 Tomcat 使用这个额外的 jar 文件，必须重启 Tomcat 实例。

范例应用

Tomcat 中的 `/examples` 应用中带有一个使用该资源工厂的范例。可以通过“JSP 范例”的链接来访问它。实际发送邮件的 servlet 的源代码则位于 `/WEB-INF/classes/SendMailServlet.java` 中。

警告：默认配置在 `localhost` 的 端口 25 上的 SMTP 服务器。如果实际情况不符，则需要编辑该 Web 应用的 `<Context>` 元素，将 `mail.smtp.host` 参数的值修改为你的网络上的 SMTP 服务器的主机名。

JDBC 数据源

简介

许多 Web 应用都需要 JDBC 驱动来访问数据库，以便能够支持该应用所需要的功能。Java EE 平台规范要求 Java EE 应用服务器针对该需求提供一个 DataSource 实现（也就是说，用于 JDBC 连接的连接池）。Tomcat 就能提供同样的支持，因此在 Tomcat 上，由于使用了这种服务，基于数据库的应用可以不用修改就能移植到任何 Java EE 服务器上运行。

要想详细了解 JDBC，可以参考以下网站或信息来源：

- http://www.oracle.com/technetwork/java/javase/jdbc/index.html 一个 Java 数据库连接性能相关信息网站。
- http://java.sun.com/j2se/1.3/docs/guide/jdbc/spec2/jdbc2.1.frame.html JDBC 2.1 API 规范。
- http://java.sun.com/products/jdbc/jdbc20.stdext.pdf JDBC 2.0 标准扩展 API（包括 `javax.sql.DataSource` API）。该包现在被称为“JDBC 可选包”。
- http://www.oracle.com/technetwork/java/javaee/overview/index.htm Java EE 平台规范（介绍了所有 Java EE 平台必须为应用提供的 JDBC 附加功能）。

注意：Tomcat 默认所支持的数据源是基于 Commons 项目 的 **DBCP** 连接池。但也可以通过编写自定义的资源工厂，使用其他实现了 `javax.sql.DataSource` 的连接池，详见下文。

安装 **JDBC** 驱动

使用 JDBC 数据源的 JNDI 资源工厂需要一个适合的 JDBC 驱动，要求它既能被 Tomcat 内部类所使用，也能被你的 Web 应用所使用。这很容易实现，只需将驱动的 JAR 文件（或多个文件）安装到 `$CATALINA_HOME/lib` 目录中即可，这样资源工厂和应用就都能使用了这一驱动了。

声明资源需求

下一步，修改 Web 应用的部署描述符文件（`/WEB-INF/web.xml`），声明 JNDI 名称以便借此查找预配置的数据源。按照惯例，所有这样的名称都应该在 `jdbc` 子上下文中声明（这个“子”是相对于标准的 `java:comp/env` 环境命名上下文而言的。`java:comp/env` 环境命名上下文是所有资源工厂的根引用）。典型的 `web.xml` 文件应如下所示：

```
<resource-ref>
  <description>
    Resource reference to a factory for java.sql.Connection
    instances that may be used for talking to a particular
    database that is configured in the <Context>
    configurartion for the web application.
  </description>
  <res-ref-name>
    jdbc/EmployeeDB
  </res-ref-name>
  <res-type>
    javax.sql.DataSource
  </res-type>
  <res-auth>
    Container
  </res-auth>
</resource-ref>
```

警告：一定要遵从 Web 应用部署描述符文件中 DTD 所需要的元素顺序。关于这点，可参看Servlet 规范中的解释。

使用资源

资源引用的典型用例如下所示：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
DataSource ds = (DataSource)
  envCtx.lookup("jdbc/EmployeeDB");

Connection conn = ds.getConnection();
... use this connection to access the database ...
conn.close();
```

注意，该应用所用的资源引用名与 Web 应用部署符中声明的完全相同。这是与下文会讲到的 `<Context>` 元素里所配置的资源工厂相匹配的。

配置 **Tomcat** 资源工厂

为了配置 Tomcat 的资源工厂，在 `<Context>` 元素中添加以下元素：

```
<Context ...>
  ...
  <Resource name="jdbc/EmployeeDB"
            auth="Container"
            type="javax.sql.DataSource"
            username="dbusername"
            password="dbpassword"
            driverClassName="org.hsql.jdbcDriver"
            url="jdbc:HypersonicSQL:database"
            maxTotal="8"
            maxIdle="4"/>
  ...
</Context>
```

注意上述代码中的资源名（这里是 `jdbc/EmployeeDB`）必须跟 Web 应用部署描述符文件中指定的值相同。

该例假定使用的是 HypersonicSQL 数据库 JDBC 驱动。可自定义 `driverClassName` 和 `driverName` 参数，使其匹配实际数据库的 JDBC 驱动与连接 URL。

Tomcat 标准数据源资源工厂（`org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory`）的配置属性如下：

- **driverClassName** 所用的 JDBC 驱动的完全合格的类名。
- **username** JDBC 驱动所要接受的数据库用户名。
- **password** JDBC 驱动所要接受的数据库密码。
- **url** 传入 JDBC 驱动的连接 URL（为了向后兼容性考虑，也同样认可 `driverName` 属性，即与之等同）。
- **initialSize** 连接池初始化过程中所创建的初始连接的数目。默认为 0。
- **maxTotal** 连接池同时所能分配的最大连接数。默认为 8。
- **minIdle** 连接池中同时空闲的最少连接数。默认为 0。
- **maxIdle** 连接池中同时空闲的最多连接数。默认为 8。
- **maxWaitMillis** 在抛出异常前，连接池等待（没有可用的连接）连接返回的最长等待毫秒数。默认为 -1（无限长时间）。

还有一些额外的用来验证连接的属性，如下所示：

- **validationQuery** 在连接返回应用之前，连接池用于验证连接的 SQL 查询。如果指定了该属性值，查询必须是一个至少能返回一行的 SQL SELECT 语句。
- **validationQueryTimeout** 验证查询返回的超时时间。默认为 -1（无限长时间）。
- **testOnBorrow** 布尔值，true 或 false，针对每次从连接池中借出的连接，判断是否应用验证查询对其验证。默认：true。
- **testOnReturn** 布尔值，true 或 false，针对每次归还给连接池的连接，判断是否应用验证查询对其验证。默认：false。

可选的 evictor thread 会清除空闲较长时间的连接，从而缩小连接池。evictor thread 不受 `minIdle` 属性值的空闲。注意，如果你只想通过配置的 `minIdle` 属性来缩小连接池，那么不需要使用 evictor thread。

默认 evictor 是禁用的，另外，可以使用下列属性来配置它：

- `timeBetweenEvictionRunsMillis` evictor 线程连续运行之间的毫秒数。默认为 -1（禁止）
- `numTestsPerEvictionRun` evictor 每次运行中，evictor 实施的用来检测空闲与否的连接数目。默认为 3。
- `minEvictableIdleTimeMillis` evictor 从连接池中清除某连接后的空闲时间，以毫秒计，默认为 30 * 60 * 1000（30分钟）。
- `testWhileIdle` 布尔值，true 或 false。对于在连接池中处于空闲状态的连接，是否应被 evictor 线程通过验证查询来验证。默认为false。

另一个可选特性是对废弃连接的移除。如果应用很久都不把某个连接返回给连接池，那么该连接就被称为废弃连接。连接池就会自动关闭这样的连接，并将其从池中移除。这么做是为了防止应用泄露连接。

默认是禁止废弃连接的，可以通过下列属性来配置：

- **removeAbandoned** 布尔值，true 或 false。确定是否去除连接池中的废弃连接。默认为 false。
- **removeAbandonedTimeout** 经过多少秒后，借用的连接可以认为被废弃。默认为 300。
- **logAbandoned** 布尔值，true 或 false。确定是否需要针对废弃了语句或连接的应用代码来记录堆栈跟踪。如果记录的话，将会带来很大的开销。默认为 false。

最后再介绍一些可以对连接池行为进行进一步微调的属性：

- **defaultAutoCommit** 布尔值，`true` 或 `false`。由连接池所创建的连接的默认自动提交状态。默认为 true。
- **defaultReadOnly** 布尔值，`true` 或 `false`。由连接池所创建的连接的默认只读状态。默认为 false。
- **defaultTransactionIsolation** 设定默认的事务隔离级别。可取值为：`NONE`、`READ_COMMITTED`、`READ_UNCOMMITTED`、`REPEATABLE_READ` 与 `SERIALIZABLE`。没有默认设置。
- **poolPreparedStatements** 布尔值，true 或 false。 是否池化 PreparedStatements 和 CallableStatements。默认为 `false`。
- **maxOpenPreparedStatements** 同时能被语句池分配的开放语句的最大数目。默认为 -1（无限）
- **defaultCatalog** catalog 默认值。默认值：未设定。
- **connectionInitSqls** 连接建立后运行的一系列 SQL 语句。各个语句间用分号（`;`）进行分隔。默认为：没有语句。
- **connectionInitSqls** 传入驱动用于创建连接的驱动特定属性。每一属性都以 `name = value` 的形式给出，多个属性用分号（`;`）进行分隔。默认：没有属性。
- **accessToUnderlyingConnectionAllowed** 布尔值，`true` 或 `false`。 是否可访问底层连接。默认为 `false`。

要想更详细地了解这些属性，请参阅 commons-dbcp 文档。

添加自定义资源工厂

如果标准资源工厂无法满足你的需求，你还可以自己编写资源工厂，然后将其集成到 Tomcat 中，在 Web 应用的 `<Context>` 元素中配置该工厂的使用方式。在下面的范例中，我们将创建一个资源工厂，只懂得如何 `com.mycompany.MyBean` bean

编写资源工厂类

你必须编写一个类来实现 JNDI 服务提供者 `javax.naming.spi.ObjectFactory` 接口。每次 Web 应用在绑定到该工厂（假设该工厂配置中，`singleton = "false"`）的上下文项上调用 `lookup()` 时，就会调用 `getObjectInstance()` 方法，该方法有如下这些参数：

- `Object obj`

创建一个能够生成 `MyBean` 实例的资源工厂，需要像下面这样来创建类：

```
package com.mycompany;

import java.util.Enumeration;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.Name;
import javax.naming.NamingException;
```

```
import javax.naming.RefAddr;
import javax.naming.Reference;
import javax.naming.spi.ObjectFactory;

public class MyBeanFactory implements ObjectFactory {

  public Object getObjectInstance(Object obj,
      Name name, Context nameCtx, Hashtable environment)
      throws NamingException {

      // Acquire an instance of our specified bean class
      MyBean bean = new MyBean();

      // Customize the bean properties from our attributes
      Reference ref = (Reference) obj;
      Enumeration addrs = ref.getAll();
      while (addrs.hasMoreElements()) {
          RefAddr addr = (RefAddr) addrs.nextElement();
          String name = addr.getType();
          String value = (String) addr.getContent();
          if (name.equals("foo")) {
              bean.setFoo(value);
          } else if (name.equals("bar")) {
              try {
                  bean.setBar(Integer.parseInt(value));
              } catch (NumberFormatException e) {
                  throw new NamingException("Invalid 'bar' value " + value);
              }
          }
      }

      // Return the customized instance
      return (bean);

  }

}
```

```
// Acquire an instance of our specified bean class 需要我们所指定的bean 类的一个实例

// Customize the bean properties from our attributes 从属性中自定义 bean 属性。

// Return the customized instance 返回自定义实例
```

在上例中，无条件地创建了 `com.mycompany.MyBean` 类的一个新实例， 并根据工厂配置中的 `<ResourceParams>` 元素（下文详述）包括的参数来填充这一实例。你应该记住，必须忽略任何名为 `factory` 的参数——参数应该用来指定工厂类自身的名字（`com.mycompany.MyBeanFactory`），而不是配置的 bean 属性。

关于 `ObjectFactory` 的更多信息，可参见 JNDI 服务提供者接口（SPI）规范。

首先参照一个 `$CATALINA_HOME/lib` 目录中包含所有 JAR 文件的类路径来编译该类。完成之后，将这个工厂类以及相应的 Bean 类解压缩到 `$CATALINA_HOME/lib`，或者 `$CATALINA_HOME/lib` 内的一个 JAR 文件中。这样，所需的类文件就能被 Catalina 内部资源与 Web 应用看到了。

声明资源需求

下一步，修改 Web 应用的部署描述符文件（`/WEB-INF/web.xml`），声明 JNDI 名称以便借此请求该 bean 的新实例。最简单的方法是使用 `<resource-env-ref>` 元素，如下所示：

```
<resource-env-ref>
```

```
  <description>
    Object factory for MyBean instances.
  </description>
  <resource-env-ref-name>
    bean/MyBeanFactory
  </resource-env-ref-name>
  <resource-env-ref-type>
    com.mycompany.MyBean
  </resource-env-ref-type>
<resource-env-ref>
```

警告：一定要遵从 Web 应用部署描述符文件中 DTD 所需要的元素顺序。关于这点，可参看Servlet 规范中的解释。

使用资源

资源引用的典型用例如下所示：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");

writer.println("foo = " + bean.getFoo() + ", bar = " +
               bean.getBar());
```

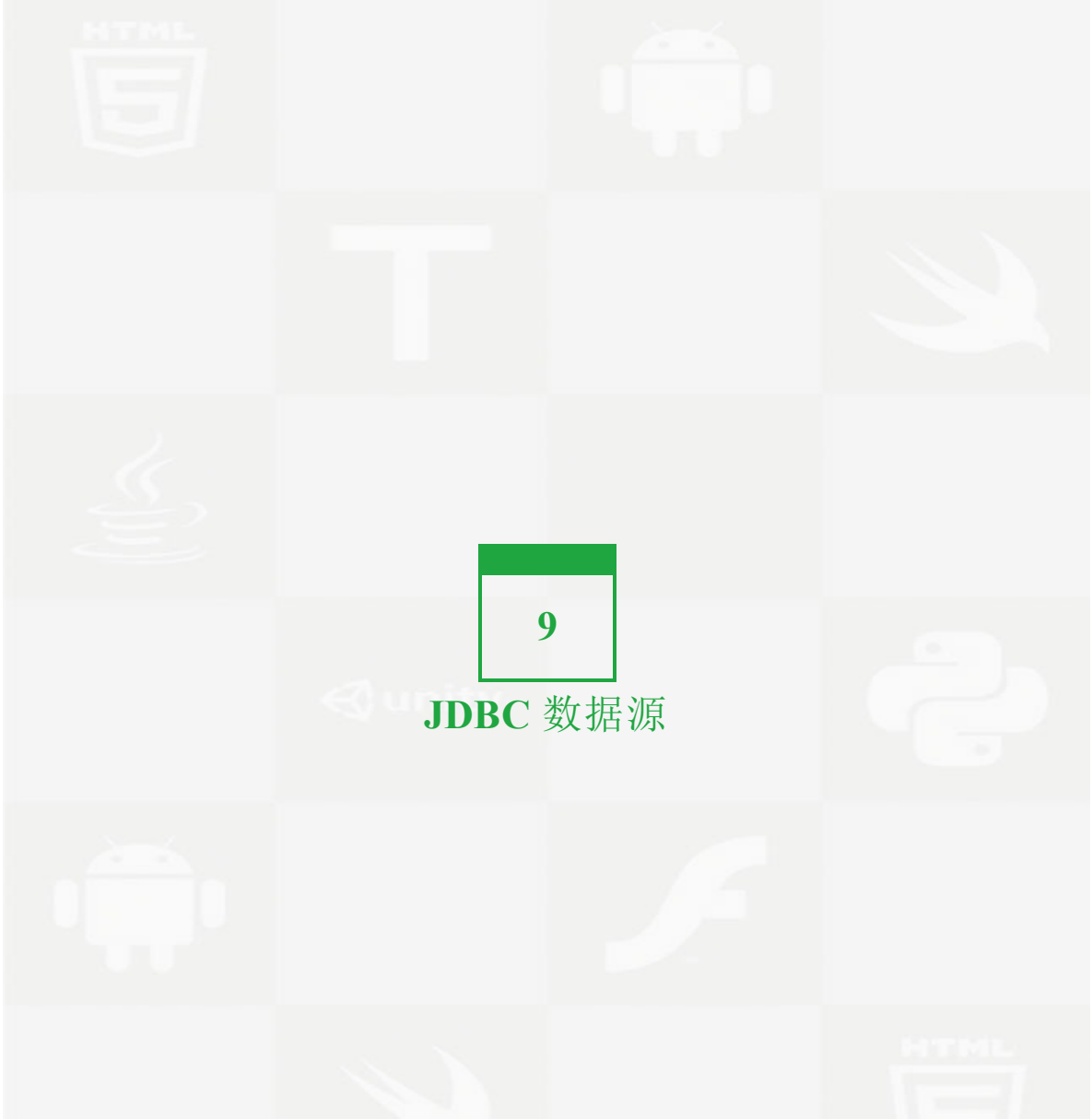
配置 **Tomcat** 资源工厂

为了配置 Tomcat 的资源工厂，在 `<Context>` 元素中添加以下元素：

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean"
            factory="com.mycompany.MyBeanFactory"
            singleton="false"
            bar="23"/>
  ...
</Context>
```

注意上述代码中的资源名（这里是 `bean/MyBeanFactory`）必须跟 Web 应用部署描述符文件中指定的值相同。另外，我们还初始化了 `bar` 属性值，从而在返回新 bean 时，导致 `setBar(23)` 被调用。由于我们没有初始化 `foo` 属性（虽然完全可以这样做），所以 bean 将含有构造函数所定义的各种默认值。

另外，你肯定能注意到，从应用开发者的角度来看，资源环境引用的声明，以及请求新实例的编程方式，都跟通用 **JavaBean** 资源（Generic JavaBean Resources）范例所用方式如出一辙。这揭示了使用 JNDI 资源封装功能的一个优点：只要维持兼容的 API，无需修改使用资源的应用，只需改变底层实现。



9

JDBC 数据源

概述

JNDI 数据源配置的相关内容已经在 JNDI 资源文档中详细介绍过。但从 Tomcat 用户的反馈意见来看，有些配置的细节问题非常棘手。

针对常用的数据库，我们已经给 Tomcat 用户提供了一些配置范例，以及关于数据库使用的一些通用技巧。本章就将展示这些范例和技巧。

另外，虽然有些注意事项来自于用户所提供的配置和反馈信息，但你可能也有不同的实践。如果经过试验，你发现某些配置可能具有广泛的助益作用，或者你觉得它们会使本章内容更加完善，请务必不吝赐教。

请注意，对比 **Tomcat 7.x** 和 **Tomcat 8.x**，**JNDI** 资源配置多少有些不同，这是因为使用的 **Apache Commons DBCP** 库的版本不同所致。所以，为了在 Tomcat 8 中使用，你最好修改老版本的 JNDI 资源配置，以便能够匹配下文范例中的格式。详情可参看Tomcat 迁移文档。

另外还要提示的是，一般来说（特别是对于本教程而言），JNDI 数据源配置会假定你已经理解了 Context 与 Host 的配置偏好，其中包括在后者配置偏好中的应用自动部署的相关内容。

DriverManager，服务提供者机制以及内存泄露

`java.sql.DriverManager` 支持服务提供者机制。这项功能的实际作用在于：对于所有可用的 JDBC 驱动，只要它们声明提供 `META-INF/services/java.sql.Driver` 文件，就会被自动发现、加载并注册，从而减轻了我们在创建 JDBC 连接之前还需要显式地加载数据库驱动的负担。但在 servlet 容器环境的所有 Java 版本中，却根本没法实现这种功能。问题在于 `java.sql.DriverManager` 只会扫描一次驱动。

Tomcat 自带的阻止 JRE 内存泄漏侦听器可以在一定程度上解决这个问题，它会在 Tomcat 启动时触发驱动扫描。该侦听器默认是启用的。只有可见于该侦听器的库（比如 `$CATALINA_BASE/lib` 中的库）才能被数据库驱动所扫描。如果你想禁用该功能，那么一定要记住：首先使用 JDBC 的 Web 应用会触发扫描，从而当该应用重新加载时会出错；对于其他依赖该功能的 Web 应用来说也会导致出错。

所以，假如应用的 `WEB-INF/lib` 目录中存在数据库驱动，那么这些应用就不能依赖服务提供者机制，而应该显式地注册驱动。

`java.sql.DriverManager` 中的驱动已经被认为是内存泄露之源。当 Web 应用停止运行时，它所注册的任何驱动都必须重新注册。当 Web 应用停止运行时，Tomcat 会尝试自动寻找并重新注册任何由 Web 应用类加载器所加载的 JDBC 驱动。但最好是由应用通过 `ServletContextListener` 来实现这一点。

数据库连接池（DBCP 2）配置

Apache Tomcat 的默认数据库连接池实现基于的是 Apache Commons 项目的库，具体来说是这两个库：

- Commons DBCP
- Commons Pool

这两个库都位于一个 JAR 文件中：`$CATALINA_HOME/lib/tomcat-dbcp.jar`。但该文件只包括连接池所需要的类，包名也已经改变了，以避免与应用冲突。

DBCP 2.0 支持 JDBC 4.1。

安装

可参阅 DBCP 文档了解完整的配置参数。

防止数据库连接池泄露

数据库连接池创建并管理着一些与数据库的连接。与打开新的连接相比，回收或重用现有的数据库连接要更为高效一些。

连接池化还存在一个问题。Web 应用必须明确地关闭 ResultSet、Statement，以及 Connection。假如 Web 应用无法关闭这些资源时，会导致这些资源再也无法被重用，从而造成了数据库连接池“泄露”。如果再也没有可用连接时，最终这将导致 Web 应用数据库连接失败。

针对该问题，有一个解决办法：通过配置 Apache Commons DBCP，记录并恢复这些废弃的数据库连接。它不仅能恢复这些连接，而且还能针对打开这些连接而又永远不关闭它们的代码生成堆栈跟踪。

为了配置 DBCP 数据源来移除并回收废弃的数据库连接，将下列属性（一个或全部）添加到你的 DBCP 数据源中的 `Resource` 配置中：

`removeAbandonedOnBorrow=true`

`removeAbandonedOnMaintenance=true`

以上属性默认都为 `false`。注意，只有当 `timeBetweenEvictionRunsMillis` 为正值，从而启用池维护时，`removeAbandonedOnMaintenance` 才能生效。关于这些属性的详情，可查看 DBCP 文档 。

使用 `removeAbandonedTimeout` 属性设置某个数据库连接闲置的秒数，超过此时段，即被认为是废弃连接。

`removeAbandonedTimeout="60"`

默认的去除废弃连接的超时为 300 秒。

将 `logAbandoned` 设为 `true`，可以让 DBCP 针对那些抛弃数据库连接资源的代码，记录堆栈跟踪信息。

`logAbandoned="true"`

默认为 `false`。

MySQL DBCP 范例

0. 简介

已报告的能够正常运作的 MySQL 与 JDBC 驱动的版本号为：

- MySQL 3.23.47、使用 InnoDB 的 MySQL 3.23.47、MySQL 3.23.58 以及 MySQL 4.0.1 alpha
- Connector/J 3.0.11-stable （JDBC 官方驱动）
- mm.mysql 2.0.14 （一个较老的 JDBC 第三方驱动）

在继续下一步的操作之前，千万不要忘了将 JDBC 驱动的 JAR 文件复制到 `$CATALINA_HOME/lib` 中。

1. MySQL配置

一定要按照下面的说明去操作，否则会出现问题。

创建一个新的测试用户、一个新的数据库，以及一张新的测试表。必须为 MySQL 用户指定一个密码。如果密码为空，那么在连接时，就会无法正常驱动。

```
mysql> GRANT ALL PRIVILEGES ON *.* TO javauser@localhost
    ->   IDENTIFIED BY 'javadude' WITH GRANT OPTION;
mysql> create database javatest;
mysql> use javatest;
mysql> create table testdata (
    ->   id int not null auto_increment primary key,
    ->   foo varchar(25),
    ->   bar int);
```

> 注意：一旦测试结束，就该把上例中的这个用户删除！

下面在 `testdata` 表中插入一些测试数据：

```
mysql> insert into testdata values(null, 'hello', 12345);
Query OK, 1 row affected (0.00 sec)

mysql> select * from testdata;
+----+-------+-------+
| ID | FOO   | BAR   |
+----+-------+-------+
|  1 | hello | 12345 |
+----+-------+-------+
1 row in set (0.00 sec)

mysql>
```

2. 上下文配置

在 Context 中添加资源声明，以便在 Tomcat 中配置 JNDI 数据源。

范例如下：

```
<Context>

    <!-- maxTotal: Maximum number of database connections in pool. Make sure you
         configure your mysqld max_connections large enough to handle
         all of your db connections. Set to -1 for no limit.
         -->

    <!-- maxIdle: Maximum number of idle database connections to retain in pool.
         Set to -1 for no limit.  See also the DBCP documentation on this
         and the minEvictableIdleTimeMillis configuration parameter.
         -->

    <!-- maxWaitMillis: Maximum time to wait for a database connection to become
```

```
available
         in ms, in this example 10 seconds. An Exception is thrown if
         this timeout is exceeded.  Set to -1 to wait indefinitely.
         -->

    <!-- username and password: MySQL username and password for database
connections  -->

    <!-- driverClassName: Class name for the old mm.mysql JDBC driver is
         org.gjt.mm.mysql.Driver - we recommend using Connector/J though.
         Class name for the official MySQL Connector/J driver is
com.mysql.jdbc.Driver.
         -->

    <!-- url: The JDBC connection url for connecting to your MySQL database.
         -->

  <Resource name="jdbc/TestDB" auth="Container" type="javax.sql.DataSource"
               maxTotal="100" maxIdle="30" maxWaitMillis="10000"
               username="javauser" password="javadude"
driverClassName="com.mysql.jdbc.Driver"
               url="jdbc:mysql://localhost:3306/javatest"/>

</Context>
```

3. web.xml 配置

为该测试应用创建一个 `WEB-INF/web.xml` 文件：

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <description>MySQL Test App</description>
  <resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/TestDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

4. 测试代码

创建一个简单的 `test.jsp` 页面，稍后将用到它。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<sql:query var="rs" dataSource="jdbc/TestDB">
select id, foo, bar from testdata
</sql:query>

<html>
  <head>
    <title>DB Test</title>
  </head>
  <body>
```

```
  <h2>Results</h2>

<c:forEach var="row" items="${rs.rows}">
    Foo ${row.foo}<br/>
    Bar ${row.bar}<br/>
</c:forEach>

  </body>
</html>
```

JSP 页面用到了 JSTL 的 SQL 和 Core taglibs。你可以从 Apache Tomcat Taglibs - Standard Tag Library 项目中获取它，不过要注意应该是 1.1.x 或之后的版本。下载 JSTL 后，将 `jstl.jar` 和 `standard.jar` 复制到 Web 应用的 `WEB-INF/lib` 目录中。

最后，将你的应用部署到 `$CATALINA_BASE/webapps`，可以采用两种方式：或者将应用以名叫 `DBTest.war` 的 WAR 文件形式部署；或者把应用放入一个叫 `DBTest` 的子目录中。

部署完毕后，就可以在浏览器输入 `http://localhost:8080/DBTest/test.jsp`，查看你的第一个劳动成果了。

Oracle 8i、9i 与 10g

0. 简介

Oracle 需要的配置和 MySQL 差不多，只不过也存在一些常见问题。

针对过去版本的 Oracle 的驱动可能以 *.zip 格式（而不是 *.jar 格式）进行分发的。Tomcat 只使用 `*.jar` 文件，而且它们还必须安装在 `$CATALINA_HOME/lib` 中。因此，`classes111.zip` 或 `classes12.zip` 这样的文件后缀应该改成 `.jar`。因为 jar 文件本来就是一种 zip 文件，因此不需要将原 zip 文件解压缩然后创建相应的 jar 文件，只需改换后缀名即可。

对于 Oracle 9i 之后的版本，应该使用 `oracle.jdbc.OracleDriver` 而不是 `oracle.jdbc.driver.OracleDriver`，因为 Oracle 规定开始弃用 `oracle.jdbc.driver.OracleDriver`，下一个重大版本将不再支持这一驱动类。

1. 上下文配置

跟前文 MySql 的配置一样，你也需要在 Context 中定义数据源。下面定义一个叫做 myoracle 的数据源，使用上文说的短驱动来连接（用户名为 scott，密码为 tiger）到名为 mysid 的SID（Oracle 系统ID，标识一个数据库的唯一标示符）。 用户 scott 使用的 Schema 就是默认的 schema。

使用 OCI 驱动，只需在 URL 字符串中将 thin 变为 oci 即可。

```
<Resource name="jdbc/myoracle" auth="Container"
              type="javax.sql.DataSource"
driverClassName="oracle.jdbc.OracleDriver"
              url="jdbc:oracle:thin:@127.0.0.1:1521:mysid"
              username="scott" password="tiger" maxTotal="20" maxIdle="10"
              maxWaitMillis="-1"/>
```

2. web.xml 配置

在创建 Web 应用的 web.xml 文件时，一定要遵从 Web 应用部署描述符文件中 DTD 所需要的元素顺序。

```
<resource-ref>
 <description>Oracle Datasource example</description>
 <res-ref-name>jdbc/myoracle</res-ref-name>
```

```
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

3. 代码范例

可以使用上文所列的范例应用（假如你创建了所需的 DB 实例和表，等等），将数据源代码用下面的代码替换：

```
Context initContext = new InitialContext();
Context envContext  = (Context)initContext.lookup("java:/comp/env");
DataSource ds = (DataSource)envContext.lookup("jdbc/myoracle");
Connection conn = ds.getConnection();
//etc.
```

PostgreSQL

0. 简介

PostgreSQL 配置与 Oracle 基本相似。

1. 所需文件

将 Postgres 的 JDBC jar 文件复制到 `$CATALINA_HOME/lib` 中。和 Oracle 配置一样，jar 文件必须放在这个目录中，DBCP 类加载器才能找到它们。不管接下来如何配置，这是首先必须要做的。

2. 资源配置

目前有两种选择：定义一个能够被 Tomcat 所有应用所共享的数据源，或者定义只能被单个应用所使用的数据源。

2a. 共享数据源配置
如果想定义能够被多个 Tomcat 应用所共享的数据源，或者只想在文件中定义自己的数据源，则采用如下配置：

尽管有些用户反馈说这样可行，但本文档作者却没有成功，希望有人能阐述清楚。

```
<Resource name="jdbc/postgres" auth="Container"
          type="javax.sql.DataSource" driverClassName="org.postgresql.Driver"
          url="jdbc:postgresql://127.0.0.1:5432/mydb"
          username="myuser" password="mypasswd" maxTotal="20" maxIdle="10"
maxWaitMillis="-1"/>
```

2b. 应用专属的资源配置
如果希望专门为某一应用定义数据源，其他 Tomcat 应用无法使用，可以使用如下配置。这种方法对 Tomcat 安装的损害性要小一些。

在你的应用的 Context 中创建一个资源定义，如下所示：

```
<Context>

<Resource name="jdbc/postgres" auth="Container"
          type="javax.sql.DataSource" driverClassName="org.postgresql.Driver"
          url="jdbc:postgresql://127.0.0.1:5432/mydb"
          username="myuser" password="mypasswd" maxTotal="20" maxIdle="10"
maxWaitMillis="-1"/>
</Context>
```

3. web.xml 配置

```
<resource-ref>
 <description>postgreSQL Datasource example</description>
 <res-ref-name>jdbc/postgres</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

4. 访问数据库

在利用程序访问数据库时，记住把 `java:/comp/env` 放在你的 JNDI lookup 方法参数的前部，如下面这段代码所示。另外，可以用任何你想用的值来替换 `jdbc/postgres`，不过记得也要用同样的值来修改上面的资源定义文件。

```
InitialContext cxt = new InitialContext();
if ( cxt == null ) {
   throw new Exception("Uh oh -- no context!");
}

DataSource ds = (DataSource) cxt.lookup( "java:/comp/env/jdbc/postgres" );

if ( ds == null ) {
   throw new Exception("Data source not found!");
}
```

非 **DBCP** 的解决方案

这些方案或者使用一个单独的数据库连接（建议仅作测试用！），或者使用其他一些池化技术。

Oracle 8i 与 OCI 客户端

简介

虽然并不能严格地解决如何使用 OCI 客户端来创建 JNDI 数据源的问题，但这些注意事项却能和上文提到的 Oracle 与 DBCP 解决方案结合起来使用。

为了使用 OCI 驱动，应该先安装一个 Oracle 客户。你应该已经通过光盘安装好了 Oracle 8i（8.1.7）客户端，并从 otn.oracle.com 下载了适用的 JDBC/OCI 驱动（Oracle8i 8.1.7.1 JDBC/OCI 驱动）。

将 `classes12.zip` 重命名为 `classes12.jar` 后，将其复制到 `$CATALINA_HOME/lib` 中。根据 Tomcat 的版本以及你所使用的 JDK，你可能还必须该文件中的删除 `javax.sql.*` 类。

连接起来

确保在 `$PATH` 或 `LD_LIBRARY PATH`（可能在 `$ORAHOME\bin`）目录下存在 `ocijdbc8.dll` 或 `.so` 文件，另外还要确认能否使用 `System.loadLibrary("ocijdbc8");` 这样的简单测试程序加载本地库。

下面你应该创建一个简单测试用 servlet 或 jsp，其中应该包含以下关键代码：

```
DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
conn =
DriverManager.getConnection("jdbc:oracle:oci8:@database","username","password");
```

目前数据库是 `host:port:SID` 形式，如果你试图访问测试用servlet/jsp，那么你会得到一个 `ServletException` 异常，造成异常的根本原因在于 `java.lang.UnsatisfiedLinkError:get_env_handle`。

分析一下，首先 `UnsatisfiedLinkError` 表明：

- JDBC 类文件和 Oracle 客户端版本不匹配。消息中透露出的意思是没有找到需要的库文件。比如，你可能使用 Oracle 8.1.6 的 class12.zip 文件，而 Oracle 客户端版本则是 8.1.5。classeXXXs.zip 文件必须与 Oracle 客户端文件版本相一致。
- 出现了一个 `$PATH, LD_LIBRARY_PATH` 问题。
- 有报告称，忽略从 otn 网站下载的驱动，使用 `$ORAHOME\jdbc\lib` 目录中的 class12.zip 文件，同样能够正常运作。

接下来，你可能还会遇到另一个错误消息：`ORA-06401 NETCMN: invalid driver designator`。

Oracle 文档是这么说的：“异常原因：登录（连接）字符串包含一个不合法的驱动标识符。解决方法：修改字符串，重新提交。”所以，如下面这样来修改数据库（`host:port:SID`）连接字符串：

```
(description=(address=(host=myhost)(protocol=tcp)(port=1521))(connect_data=
(sid=orcl)))
```

常见问题

下面是一些 Web 应用在使用数据库时经常会遇到的问题，以及一些应对技巧。

数据库连接间歇性失败

Tomcat 运行在 JVM 中。JVM 周期性地会执行垃圾回收（GC），清除不再使用的 Java 对象。当 JVM 执行 GC 时，Tomcat 中的代码执行就会终止。如果配置好的数据库连接建立的最长时间小于垃圾回收的时间，数据库连接就会失败。

在启动 Tomcat 时，将 `-verbose:gc` 参数添加到 `CATALINA_OPTS` 环境变量中，就能知道垃圾回收所占用的时间了。在启用 `verbose:gc` 后，`$CATALINA_BASE/logs/catalina.out` 日志文件就能包含每次垃圾回收的数据，其中也包括它所占用的时间。

正确调整 JVM 后，垃圾回收可以做到在 99% 的情况下占用时间不超过 1 秒。剩余的情况则只占用几秒钟的时间，只有极少数情况下 GC 会占用超过 10 秒钟的时间。

保证让数据库连接超时设定在 10~15 秒。对于 DBCP，可以使用 `maxWaitMillis` 参数来设置。

随机性的连接关闭异常

当某一请求从连接池中获取了一个数据库连接，然后关闭了它两次时，往往会出现这样的异常消息。使用连接池时，关闭连接，就会把它归还给连接池，以便之后其他的请求能够重用该连接，而并不会关闭连接。Tomcat 使用多个线程来处理并发请求。下面这个范例就演示了，在 Tomcat 中，一系列事件导致了这种错误。

运行在线程 **1** 中的请求 **1** 获取了一个连接。

请求 **1** 关闭了数据库连接。

JVM 将运行的线程切换为线程 **2**。

线程 **2** 中运行的请求 **2** 获取了一个数据库连接。
（同一个数据库连接刚被请求 **1** 关闭）

JVM 又将运行的线程切换回为线程 **1**。

请求 **1** 第二次关闭了数据库连接。

JVM 将运行的线程切换回线程 **2**。

请求 **2** 和线程 **2** 试图使用数据库连接，但却失败了。因为请求 **1** 已经关闭了它。

```
Connection conn = null;
Statement stmt = null;  // Or PreparedStatement if needed
ResultSet rs = null;
try {
  conn = ... get connection from connection pool ...
  stmt = conn.createStatement("select ...");
  rs = stmt.executeQuery();
  ... iterate through the result set ...
  rs.close();
  rs = null;
  stmt.close();
  stmt = null;
  conn.close(); // Return to connection pool
  conn = null;  // Make sure we don't close it twice
```

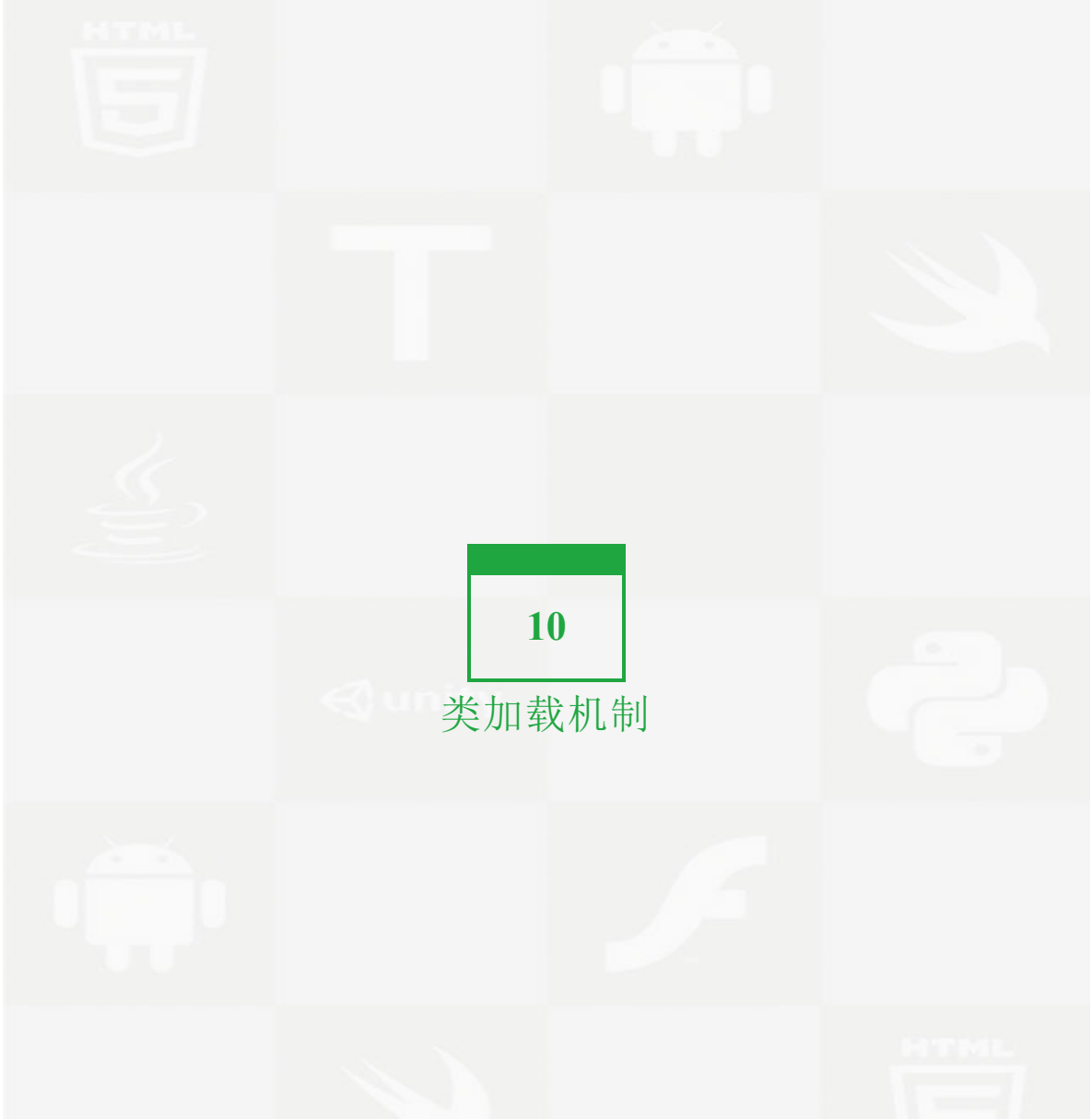
```
} catch (SQLException e) {
  ... deal with errors ...
} finally {
  // Always make sure result sets and statements are closed,
  // and the connection is returned to the pool
  if (rs != null) {
    try { rs.close(); } catch (SQLException e) { ; }
    rs = null;
  }
  if (stmt != null) {
    try { stmt.close(); } catch (SQLException e) { ; }
    stmt = null;
  }
  if (conn != null) {
    try { conn.close(); } catch (SQLException e) { ; }
    conn = null;
  }
}
```

上下文与全局命名资源

注意，虽然在上面的说明中，把 JNDI 声明放在一个 Context 元素里面，但还是有可能（而且有时更需要）把这些声明放在服务器配置文件的 GlobalNamingResources 区域。被放置在 GlobalNamingResources 区域的资源将会被服务器的各个上下文所共享。

JNDI 资源命名和 Realm 交互

为了让 Realm 能运作，realm 必须指向定义在 `<GlobalNamingResources>` 或 `<Context>` 区域中的数据源，而不是 `<ResourceLink>` 重新命名的数据源。



10

类加载机制

概述

与很多服务器应用一样，Tomcat 也安装了各种类加载器（那就是实现了 `java.lang.ClassLoader` 的类）。借助类加载器，容器的不同部分以及运行在容器上的 Web 应用就可以访问不同的仓库（保存着可使用的类和资源）。这个机制实现了 Servlet 规范 2.4 版（尤其是 9.4 节和 9.6 节）里所定义的功能。

在 Java 环境中，类加载器的布局结构是一种父子树的形式。通常，类加载器被请求加载一个特定的类或资源时，它会先把这一请求委托给它的父类加载器，只有（一个或多个）父类加载器无法找到请求的类或资源时，它才开始查看自身的仓库。注意，Web 应用的类加载器模式跟这个稍有不同，下文将详细介绍，但基本原理是一样。

当 Tomcat 启动后，它就会创建一组类加载器，这些类加载器被布局成如下图所示这种父子关系，父类加载器在子类加载器之上：

```
      Bootstrap
          |
       System
          |
       Common
       /     \
  Webapp1   Webapp2 ...
```

接下来，通过几节内容来详细说明每一个类加载器的特点，其中还将讲解这些加载器可使用的类和资源的来源。

类加载器定义

如上图所示，Tomcat 在初始化时会创建如下这些类加载器：

- **Bootstrap** 这种类加载器包含 JVM 所提供的基本的运行时类，以及来自系统扩展目录（`$JAVA_HOME/jre/lib/ext`）里 JAR 文件中的类。注意：在有些 JVM 的实现中，它的作用不仅仅是类加载器，或者它可能根本不可见（作为类加载器）。
- **System** 这种类加载器通常是根据 `CLASSPATH` 环境变量内容进行初始化的。所有的这些类对于 Tomcat 内部类以及 Web 应用来说都是可见的。不过，标准的 Tomcat 启动脚本（`$CATALINA_HOME/bin/catalina.sh` 或 `%CATALINA_HOME%\bin\catalina.bat`）完全忽略了 `CLASSPATH` 环境变量自身的内容，相反从下列仓库来构建系统类加载器：
  - `$CATALINA_HOME/bin/bootstrap.jar` 包含用来初始化 Tomcat 服务器的 `main()` 方法，以及它所依赖的类加载器实现类。
  - `$CATALINA_BASE/bin/tomcat-juli.jar` 或 `$CATALINA_HOME/bin/tomcat-juli.jar` 日志实现类。其中包括了对 `java.util.logging` API 的功能增强类（Tomcat JULI），以及对 Tomcat 内部使用的 Apache Commons 日志库的包重命名副本。详情参看 Tomcat 日志文档。

    如果 *$CATALINA_BASE/bin* 中存在 `tomcat-juli.jar`，就不会使用 *$CATALINA_HOME/bin* 中的那一个。它有助于日志的特定配置。
  - `$CATALINA_HOME/bin/commons-daemon.jar` Apache Commons Daemon 项目的类。该 JAR 文件并不存在于由 `catalina.bat` 或 `catalina.sh` 脚本所创建的 `CLASSPATH` 中，而是引用自 *bootstrap.jar* 的清单文件。
- **Common** 这种类加载器包含更多的额外类，它们对于Tomcat 内部类以及所有 Web 应用都是可见的。

  通常，应用类不会放在这里。该类加载器所搜索的位置定义在 `$CATALINA_BASE/conf/catalina.properties` 的 `common.loader` 属性中。默认的设置会搜索下列位置（按照列表中的上下顺序）。
  - `$CATALINA_BASE/lib` 中的解包的类和资源。
  - `$CATALINA_BASE/lib` 中的 JAR 文件。
  - `$CATALINA_HOME/lib` 中的解包类和资源。
  - `$CATALINA_HOME/lib` 中的 JAR 文件。

  默认，它包含以下这些内容：
  - *annotations-api.jar* JavaEE 注释类。
  - *catalina.jar* Tomcat 的 Catalina servlet 容器部分的实现。
  - *catalina-ant.jar* Tomcat Catalina Ant 任务。
  - *catalina-ha.jar* 高可用性包。
  - *catalina-storeconfig.jar*
  - *catalina-tribes.jar* 组通信包
  - *ecj-*.jar* Eclipse JDT Java 编译器
  - *el-api.jar* EL 3.0 API
  - *jasper.jar* Tomcat Jasper JSP 编译器与运行时
  - *jasper-el.jar* Tomcat Jasper EL 实现
  - *jsp-api.jar* JSP 2.3 API
  - *servlet-api.jar* Servlet 3.1 API

- *tomcat-api.jar* Tomcat 定义的一些接口
- *tomcat-coyote.jar* Tomcat 连接器与工具类。
- *tomcat-dbcp.jar* 数据库连接池实现，基于 Apache Commons Pool 的包重命名副本和 Apache Commons DBCP。
- *tomcat-i18n-**.jar* 包含其他语言资源束的可选 JAR。因为默认的资源束也可以包含在每个单独的 JAR 文件中，所以如果不需要国际化信息，可以将其安全地移除。
- *tomcat-jdbc.jar* 一个数据库连接池替代实现，又被称作 Tomcat JDBC 池。详情参看 JDBC 连接池文档。
- *tomcat-util.jar* Apache Tomcat 多种组件所使用的常用类。
- *tomcat-websocket.jar* WebSocket 1.1 实现
- *websocket-api.jar* WebSocket 1.1 API

- **WebappX** 为每个部署在单个 Tomcat 实例中的 Web 应用创建的类加载器。你的 Web 应用的 `/WEB-INF/classes` 目录中所有的解包类及资源，以及 `/WEB-INF/lib` 目录下 JAR 文件中的所有类及资源，对于该应用而言都是可见的，但对于其他应用来说则不可见。

如上所述，Web 应用类加载器背离了默认的 Java 委托模式（根据 Servlet 规范 2.4 版的 9.7.2 Web Application Classloader一节中提供的建议）。当某个请求想从 Web 应用的 WebappX 类加载器中加载类时，该类加载器会先查看自己的仓库，而不是预先进行委托处理。There are exceptions。JRE 基类的部分类不能被重写。对于一些类（比如 J2SE 1.4+ 的 XML 解析器组件），可以使用 J2SE 1.4 支持的特性。最后，类加载器会显式地忽略所有包含 Servlet API 类的 JAR 文件，所以不要在 Web 应用包含任何这样的 JAR 文件。Tomcat 其他的类加载器则遵循常用的委托模式。

因此，从 Web 应用的角度来看，加载类或资源时，要查看的仓库及其顺序如下：

- JVM 的 Bootstrap 类
- Web 应用的 `/WEB-INF/classes` 类
- Web 应用的 `/WEB-INF/lib/*.jar` 类
- System 类加载器的类（如上所述）
- Common 类加载器的类（如上所述）

如果 Web 应用类加载器配置有 `<Loader delegate="true"/>`，则顺序变为：

- JVM 的 Bootstrap 类
- System 类加载器的类（如上所述）
- Common 类加载器的类（如上所述）
- Web 应用的 `/WEB-INF/classes` 类
- Web 应用的 `/WEB-INF/lib/*.jar` 类

XML解析器和 Java

从 Java 1.4 版起，JRE 就包含了一个 JAXP API 的副本和一个 XML 解析器。这对希望使用自己的 XML 解析器的应用产生了一定的影响。

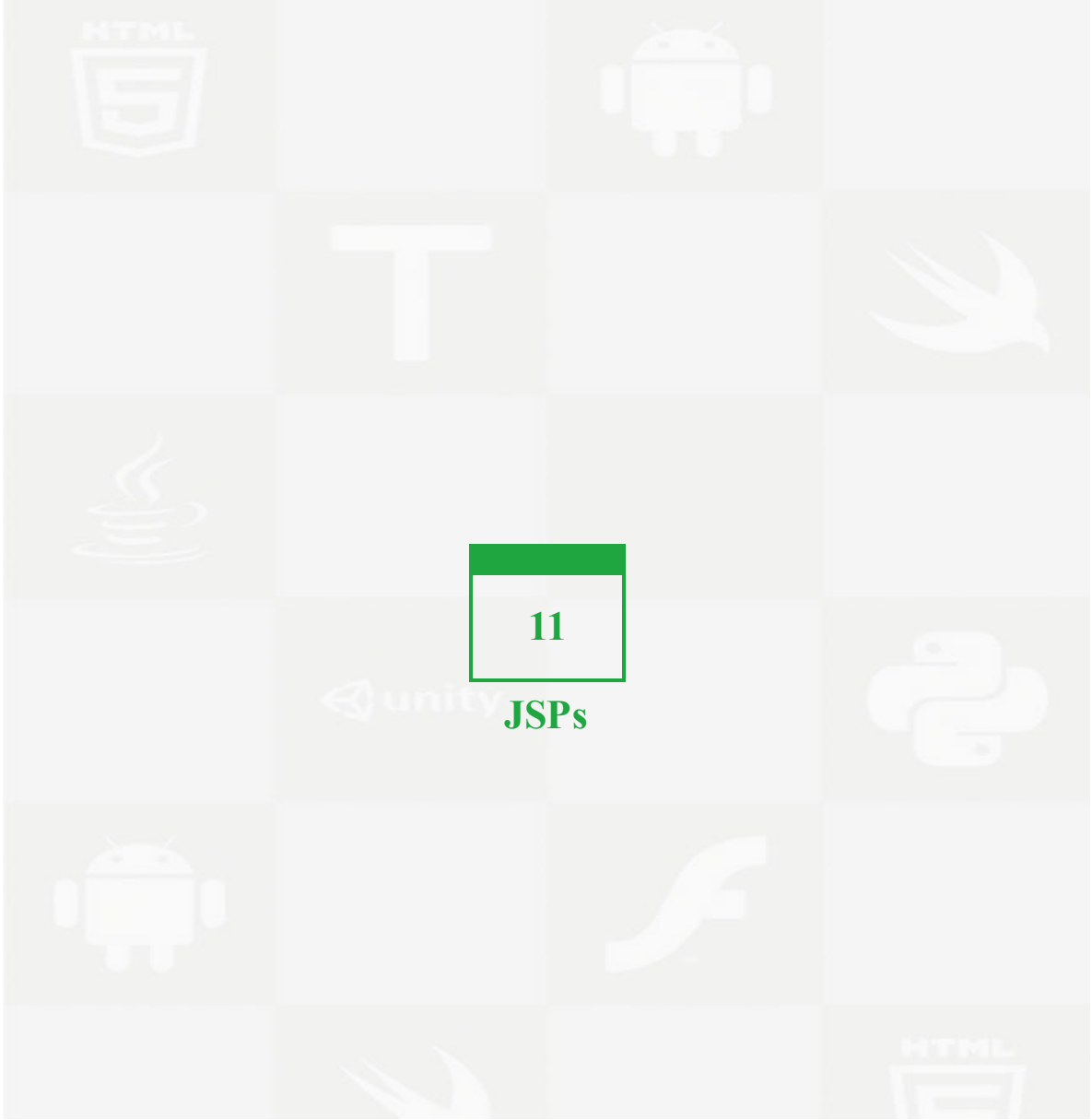
在过去的 Tomcat 中，你只需在 Tomcat 库中简单地换掉 XML 解析器，就能改变所有 Web 应用使用的解析器。但对于现在版本的 Java 而言，这一技术并没有效果，因为通常的类加载器委托进程往往会优先选择 JDK 内部的实现。

Java 支持一种叫做“授权标准覆盖机制”，从而能够替换在 JCP 之外创建的 API（例如 W3C 的 DOM 和 SAX）。它还可以用于更新 XML 解析器实现。关于此机制的详情，请参看 http://docs.oracle.com/javase/1.5.0/docs/guide/standards/index.html。

为了利用该机制，Tomcat 在启动容器的命令行中包含了系统属性设置 `-Djava.endorsed.dirs=$JAVA_ENDORSED_DIRS`。该选项的默认值为 `$CATALINA_HOME/endorsed`。但要注意，这个 `endorsed` 目录并非默认创建的。

安全管理器下运行

当在安全管理器下运行时，类被允许加载的位置也是基于策略文件中的内容，详情可查看 安全管理器文档。



11

JSPs

简介

Tomcat 8.0 使用 Jasper 2 JSP 引擎去实现 JavaServer Pages 2.3 规范。

Jasper 2 经过了重新设计，极大改善了上一版 Jasper 的性能。除了一般性的代码改进之外，还做出了以下改变：

- **JSP** 自定义标签池化 针对 JSP 自定义标签（JSP Custom Tags）实例化的 Java 对象现已可以被池化和重用，从而极大提高了使用自定义标签的 JSP 页面的性能。
- **JSP** 后台编译 如果你更改了一个已经编译的 JSP 页面，Jasper 2 会在后台重新编译该页面。之前编译的 JSP 页面将依然能够服务请求。一旦新页面编译成功，它就会自动代替旧页面。这能提高生产服务器上的 JSP 页面的可用性。
- 能够重新编译发生改动的 **JSP** 页面 Jasper 2 现在能够侦测页面何时出现改动，然后重新编译父 JSP。
- 用于编译 **JSP** 页面的 **JDT** 编译器 Eclipse JDT Java 编译器现在能用来编译 JSP java 源代码。该编译器从容器类加载器加载源代码支持。Ant 与 javac 依旧能够使用。

Jasper 可以使用 servlet 类 `org.apache.jasper.servlet.JspServlet`。

配置

Jasper 默认就是用于开发 Web 应用的。关于如何在 Tomcat 生产服务器中配置并使用 Jasper，可参考生产环境配置一节内容。

在全局性的 `$CATALINA_BASE/conf/web.xml` 中使用如下初始参数，来配置实现 Jasper 的 servlet。

- **checkInterval** 如果 `development` 为 false，并且 `checkInterval` 大于 0，则开启后台编译。`checkInterval` 参数的含义就是在检查某个 JSP 页面（以及从属文件）是否需要重新编译时，几次检查的间隔时间（以秒计）。默认为 0 秒。
- **classdebuginfo** 是否应在编译类文件时带上调试信息？布尔值，默认为 `true`。
- **classpath** 对生成的 servlet 进行编译时将要使用的类路径。如果 `ServletContext` 属性 `org.apache.jasper.Constants.SERVLET_CLASSPATH` 没有设置。当在 Tomcat 中使用 Jasper 时，该属性经常设置。默认情况下，根据当前 Web 应用，动态创建类路径。
- **compiler** 应用何种编译器 Ant 编译 JSP 页面。该参数的有效值与 Ant 的 javac 任务的编译器属性值完全相同。如果没有设置该值，则采用默认的 Eclipse JDT Java 编译器，而不是 Ant。该参数没有默认值，如果该属性被设置，则就应该使用 `setenv.[sh|bat]` 将 `ant.jar`、`ant-launcher.jar` 与 `tools.jar` 添加到 `CLASSPATH` 环境变量中。
- **compilerSourceVM** 与源文件所兼容的 JDK 版本？（默认值：`1.7`）
- **compilerTargetVM** 与生成文件所兼容的 JDK 版本？（默认值：`1.7`）
- **development** Jasper 是否被用于开发模式？如果为 true，可能通过 `modificationTestInterval` 参数来指定检查 JSP 更改情况的频率。布尔值，默认为 `true`。
- **displaySourceFragment** 异常信息是否应包含源代码片段？布尔值，默认为 `true`。
- **dumpSmap** JSR45 调试的 SMAP 信息是否应转储到一个文件？布尔值，默认为 `false`。如果 `suppressSmap` 为 `true`，则该参数值为 `false`。
- **enablePooling** 是否启用标签处理池（tag handler pooling）？这是一个编译选项。它不会影响已编译的 JSP 行为。布尔值，默认为 `true`。
- **engineOptionsClass** 允许指定用来配置 Jasper 的 Options 类。如果不存在，则使用默认的 EmbeddedServletOptions。
- **errorOnUseBeanInvalidClassAttribute** 当 useBean 行为中的类属性值不是合法的 bean 类时，Jasper 是否弹出一个错误？布尔值，默认为 `true`。
- **fork** Ant 是否应该分叉（fork）编译 JSP 页面，以便在单独的 JVM 中执行编译？布尔值，默认为 `true`。
- **genStringAsCharArray** 为了在一些情况下提高性能，是否应将文本字符串生成字符数组？默认为 `false`。
- **ieClassId** 使用 标记时，被送入 IE 浏览器中的类 ID 值。默认值为：`clsid:8AD9C840-044E-11D1-B3E9-00805F499D93`。
- **javaEncoding** 用于生成 Java 源文件的 Java 文件编码。默认为 `UTF-8`。
- **keepgenerated** 对于每个页面所生成的 Java 源代码，应该保留还是删除？布尔值，默认为 `true`（保留）。
- **mappedfile** 为便于调试，是否应该生成静态内容，每行输入都带有一个打印语句？布尔值，默认为 `true`。
- **maxLoadedJsps** Web 应用所能加载的 JSP 的最大数量。如果超出此数目，就卸载最近最少使用的 JSP，以防

止任何时刻加载的 JSP 数目不超过此限。0 或负值代表没有限制。默认为 -1。

- **jspIdleTimeout** JSP 在被卸载前，处于空闲状态的时间（以秒计）。0 或负值代表永远不会卸载。默认为 `-1`。
- **modificationTestInterval** 自上次检查 JSP 修改起，造成 JSP（以及从属文件）没有被检查修改的指定时间间隔（以秒计）。取值为 0 时，每次访问都会检查 JSP 修改。只用于开发模式下。默认为 `4` 秒。
- **recompileOnFail** 如果 JSP 编译失败，是否应该忽略 `modificationTestInterval`，下一次访问是否触发重新编译的尝试？只用在开发模式下，并且默认是禁止的，因为编译会使用大量的资源，是极其昂贵的过程。
- **scratchdir** 编译 JSP 页面时应该使用的临时目录（scratch directory）。默认为当前 Web 应用的工作目录。
- **suppressSmap** 是否禁止 JSR45 调试时生成的 SMAP 信息？`true` 或 `false`，缺省为 `false`。
- **trimSpaces** 是否应清除模板文本中行为与指令之间的的空格？默认为 `false`。
- **xpoweredBy** 是否通过生成的 servlet 添加 X-Powered-By 响应头？布尔值，默认为 `false`。

Eclipse JDT 的 Java 编译器被指定为默认的编译器。它非常先进，能够从 Tomcat 类加载器中加载所有的依赖关系。这将非常有助于编译带有几十个 JAR 文件的大型安装。在较快的服务器上，还可能实现以次秒级周期对大型 JSP 页面进行重新编译。

通过配置上述编译器属性，之前版本 Tomcat 所用的 Apache Ant 可以替代新的编译器。

已知问题

bug 39089报告指出，在编译非常大的 JSP 时，已知的 JVM 问题 bug 6294277 可能会导致出现 `java.lang.InternalError: name is too long to represent` 异常。如果出现这一问题，可以采用下列办法来解决：

- 减少 JSP 大小。
- 将 `suppressSmap` 设为 `true`，禁止生成 SMAP 信息与 JSR-045 支持。

生产配置

能做的最重要的 JSP 优化就是对 JSP 进行预编译。但这通常不太可能（比如说，使用 jsp-property-group 功能时）或者说不太实际，这种情况下，如何配置Jasper Servlet 就变得很关键了。

在生产级 Tomcat 服务器上使用 Jasper 2 时，应该考虑将默认配置进行如下这番修改：

- **development** 针对 JSP 页面编译，禁用访问检查，可以将其设为 `false`。
- **genStringAsCharArray** 设定为 `true` 可以生成稍微更有效率的字符串数组。
- **modificationTestInterval** 如果由于某种原因（如动态生成 JSP 页面），必须将 `development` 设为 `true`，提高该值将大幅改善性能。
- **trimSpaces** 设为 `true` 可以去除响应中的无用字节。

应用编译

使用 Ant 是利用 JSPC 编译 Web 应用的首选方式。注意在预编译 JSP 页面时，如果 `suppressSmap` 为 `false`，而 `compile` 为 true，则 SMAP 信息只能包含在最后的类中。使用下面的脚本来预编译 Web 应用（在 deployer 下载中也包含类似的脚本）。

```
<project name="Webapp Precompilation" default="all" basedir=".">

   <import file="${tomcat.home}/bin/catalina-tasks.xml"/>

   <target name="jspc">

    <jasper
             validateXml="false"
             uriroot="${webapp.path}"
             webXmlFragment="${webapp.path}/WEB-INF/generated_web.xml"
             outputDir="${webapp.path}/WEB-INF/src" />

  </target>

  <target name="compile">

    <mkdir dir="${webapp.path}/WEB-INF/classes"/>
    <mkdir dir="${webapp.path}/WEB-INF/lib"/>

    <javac destdir="${webapp.path}/WEB-INF/classes"
           optimize="off"
           debug="on" failonerror="false"
           srcdir="${webapp.path}/WEB-INF/src"
           excludes="**/*.smap">
      <classpath>
        <pathelement location="${webapp.path}/WEB-INF/classes"/>
        <fileset dir="${webapp.path}/WEB-INF/lib">
          <include name="*.jar"/>
        </fileset>
        <pathelement location="${tomcat.home}/lib"/>
        <fileset dir="${tomcat.home}/lib">
          <include name="*.jar"/>
        </fileset>
        <fileset dir="${tomcat.home}/bin">
          <include name="*.jar"/>
        </fileset>
      </classpath>
      <include name="**" />
      <exclude name="tags/**" />
    </javac>

  </target>

  <target name="all" depends="jspc,compile">
  </target>

  <target name="cleanup">
    <delete>
        <fileset dir="${webapp.path}/WEB-INF/src"/>
        <fileset dir="${webapp.path}/WEB-INF/classes/org/apache/jsp"/>
    </delete>
  </target>
```

```
</project>
```

下面的代码可以用来运行该脚本（利用 Tomcat 基本路径与指向应被预编译 Web 应用的路径来取代令牌）

```
$ANT_HOME/bin/ant -Dtomcat.home=<$TOMCAT_HOME> -Dwebapp.path=<$WEBAPP_PATH>
```

然后，必须在 Web 应用部署描述符文件中添加预编译过程中生成的 servlet 的声明与映射。将 `${webapp.path}/WEB-INF/generated_web.xml` 插入 `${webapp.path}/WEB-INF/web.xml` 文件中合适的位置。使用 Manager 重启 Web 应用，测试应用，以便验证应用能正常使用预编译 servlet。利用Web 应用部署描述符文件中的一个适当的令牌，也能使用 Ant 过滤功能自动插入生成的 servlet 声明与映射。这实际上就是 Tomcat 所分配的所有 Web 应用能作为构建进程中的一部分而自动编译的原理。

在 Jasper 任务中，还可以使用选项 `addWebXmlMappings`，它可以将 `${webapp.path}/WEB-INF/web.xml` 中的当前 Web 应用部署描述符文件自动与 `${webapp.path}/WEB-INF/generated web.xml` 进行合并。当你想在 JSP 页面中使用 Java 6 功能时，添加下列 javac 编译器任务属性：`source="1.6" target="1.6"`。对于动态应用而言，还可以使用 `optimize="on"` 进行编译，注意，不用带调试信息：`debug="off"`。

当首次出现 jsp 语法错误时，假如你不想停止 jsp 生成，可以使用 `failOnError="false"` 和 `showSuccess="true"`，将所有成功生成的 *jsp to java* 打印出来。这种做法有时非常有用，比如当你想要在 `${webapp.path}/WEB-INF/src` 中清除生成的 java 源文件以及 `${webapp.path}/WEB-INF/classes/org/apache/jsp` 中的编译 jsp 的 servlet 类时。

提示：

- 当你换用另一版本的 Tomcat 时，需要重新生成和编译 JSP 页面。
- 在服务器运行时使用 Java 系统属性，通过设定 `org.apache.jasper.runtime.JspFactoryImpl.USE POOL=false` 禁用 PageContext 池化，利用 `org.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true` 限制缓存。注意，改变默认值可能会影响性能，但这种情况跟具体的应用有关。

优化

Jasper 还提供了很多扩展点，能让用户针对具体的环境而优化行为。

标签插件机制就是首先要谈到的一个扩展点。对于提供给 Web 应用使用的标签处理器而言，它能提供多种替代实现。标签插件 通过位于 `WEB-INF` 的 `tagPlugins.xml` 进行注册。Jasper 本身还包含了一个 JSTL 的范例插件。

表达式语言（EL，Expression Language）解释器则是另外一个扩展点。通过 `ServletContext` 可以配置替代的 EL 解释器。可以参看 ELInterpreterFactory Java 文档来了解如何配置替代的 EL 解释器。

12

SSL/TLS 配置

Quick Start

下列说明将使用变量名 `$CATALINA_BASE` 来表示多数相对路径所基于的基本目录。如果没有为 Tomcat 多个实例设置 CATALINA_BASE 目录，则 `$CATALINA_BASE` 就会设定为 `$CATALINA_HOME` 的值，也就是你安装 Tomcat 的目录。

在 Tomcat 中安装并配置 SSL/TLS 支持，只需遵循下列几步即可。详细信息可参看文档后续介绍。

1. 创建一个 keystore 文件保存服务器的私有密钥和自签名证书：
   Windows：
   `"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA`
   UNIX：
   `$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA`

   指定密码为“changeit”。

2. 取消对 `$CATALINA_BASE/conf/server.xml` 中 “SSL HTTP/1.1 Connector” 一项的注释状态。按照下文中配置一节所描述的方式进行修改。

SSL/TLS 简介

安全传输层协议（TLS）与其前辈加密套接字（SSL）都是用于保证 Web 浏览器与 Web 服务器通过安全连接进行通信的技术。利用这些技术，我们所要传送的数据会在一端进行加密，传输到另一端后再进行解密（在处理数据之前）。这是一种双向的操作，服务器和浏览器都能在发送数据前对它们进行加密处理。

SSL/TLS 协议的另一个重要方面是认证。当我们初始通过安全连接与 Web 服务器进行通信时，服务器将提供给 Web 浏览器一组“证书”形式的凭证，用来证明站点的归属方以及站点的具体声明。某些情况下，服务器也会请求浏览器出示证书，来证明作为操作者的“你”所宣称的身份是否属实。这种证书叫做“客户端证书”，但事实上它更多地用于 B2B（企业对企业电子商务）的交易中，而并非针对个人用户。大多数启用了 SSL 协议的 Web 服务器都不需要客户端认证。

SSL/TLS 与 Tomcat

一定要注意的是，通常只有当 Tomcat 是独立运行的 Web 服务器时，才有必要去配置 Tomcat 以便利用加密套接字。具体细节可参看 Security Considerations Document。当 Tomcat 以 Servlet/JSP 容器的形式在其他 Web 服务器（比如 Apache 或 Microsoft IIS）背后运行时，通常需要配置的是主 Web 服务器，用主服务器来处理与用户的 SSL 连接。主服务器一般会利用所有的 SSL 相关功能，将任何只有 Tomcat 才能处理的请求进行解密后再传给 Tomcat。同样，Tomcat 会返回明文形式的响应，这些响应在被传输到用户浏览器之前会被主服务器进行加密处理。在这种情境下，Tomcat 知道主服务器与客户端的所有通信都是通过安全连接进行的（因为应用要求），但 Tomcat 自身无法参与到加密与解密的过程中。

证书

为了实现 SSL，Web 服务器必须对每一个接受安全连接的外部接口（IP 地址）配备相应的证书。这种设计方式的理论在于，服务器必须提供某种可信的保证（尤其对于接收敏感信息而言），保证它的拥有者是你所认为的角色。限于本章篇幅，不再赘述关于证书的详细解释，只需要把它认为成是一种 IP 地址的“数字驾照”即可。它声明了与站点相关的组织，以及一些关于站点拥有者或管理者的基本联系信息。

这种“数字驾照”的持有者对其进行了加密签名，从而使得它极难伪造。对于参与电子商务的站点或者其他一些身份验证显得非常重要的商业交易来说，证书通常都是从一些比较权威公正的 CA （ Certificate Authority，认证机构）购买的，比较知名的 CA 有 VeriSign 、Thawte 等。这些证书可经电子验证。实际上，CA 会保证所颁发证书的真实性，所以你完全可以放心。

不过，在很多情况下，验证并不是问题的关键。管理员可能只想保证服务器所传输与接收的数据是秘密的，不会被某些人通过连接来窃取。幸运的是，Java 提供了一个简单的命令行工具：`keytool`。它能轻松创建一个“自签名”的证书，这种自签名证书是由用户生成的一种证书，未经任何知名 CA 给予官方保证，因此它的真实性也无法确定。再说一次，是否认证，完全根据你的需求。

运行 SSL 通常需要注意的一些内容

当用户首次访问你站点上的安全页面时，页面通常会提供给他一个对话框，包含证书相关细节（比如组织及联系方式等），并且询问他是否愿意承认该证书为有效证书，然后再进行下一步的事务。一些浏览器可能会提供一个选项，允许永远承认给出的证书的有效性，这样就不会在用户每次访问站点时打扰他们了。但有些浏览器不会提供这种选项。一旦用户承认了证书的有效性，那么在整个的浏览器会话期间，证书都被认为是有效的。

虽然 SSL 协议的意图是尽可能有助于提供安全且高效的连接，但从性能角度来考虑，加密与解密是非茶馆耗费计算资源的，因此将整个 Web 应用都运行在 SSL 协议下是完全没有必要的，开发者需要挑选需要安全连接的页面。对于一个相当繁忙的网站来说，通常只会在特定页面上使用 SSL 协议，也就是可能交换敏感信息的页面，比如：登录页面、个人信息页面、购物车结账页面（可能会输入信用卡信息），等等。应用中的任何一个页面都可以通过加密套接字来请求访问，只需将页面地址的前缀 `http:` 换成 `https:` 即可。绝对需要安全连接的页面应该检查该页面请求所关联的协议类型，如果发现没有指定 `https:`，则采取适当行为。

最后，在安全连接上使用基于名称的虚拟主机可能会造成一定的问题。这正是 SSL 协议的局限。SSL 握手过程，即客户端浏览器接收服务器证书，必须发生在 HTTP 请求被访问前。因此包含虚拟主机名称的请求信息不能先于认证而确定，也不可能为单个 IP 地址指定多个证书。如果单个 IP 地址的所有虚拟主机都需要利用同样证书来认证的话，那么多个虚拟主机不应该干涉服务器通常的 SSL 操作。但是要注意一点，多数客户端浏览器会将服务器域名与证书中的多个域名（如果存在的话，）进行对比，如果域名出现不匹配，则浏览器会向用户显示警告信息。一般来说，生产环境中，通常只有使用基于地址的虚拟主机利用 SSL。

配置

1. 准备证书密钥存储库

Tomcat 目前只能操作 `JKS`、`PKCS11`、`PKCS12` 格式的密钥存储库。`JKS` 是 Java 标准的“Java 密钥存储库”格式，是通过 `keytool` 命令行工具创建的。该工具包含在 JDK 中。`PKCS12` 格式一种互联网标准，可以通过 OpenSSL 和 Microsoft 的 Key-Manager 来。

密钥存储库中的每一项都通过一个别名字符串来标识。尽管许多密码存储库实现都在处理别名时不区分大小写，但区分大小写的实现也是允许的。比如，`PKCS11` 规范需要别名是区分大小写的。为了避免别名大小写敏感的问题，不建议使用只有大小写不同的别名。

为了将现有的证书导入 `JKS` 密码存储库，请查阅关于 `keytool` 的相关文档（位于 JDK 文档包里）。注意，OpenSSL 经常会在密码前加上易于理解的注释，但 `keytool` 并不支持这一点。所以如果证书里的密码数据前面有注释的话，在利用 `keytool` 导入证书前，一定要清除它们。

要想把一个已有的由你自己的 CA 所签名的证书导入使用 OpenSSL 的 `PKCS12` 密码存储库，应该执行如下命令：

```
openssl pkcs12 -export -in mycert.crt -inkey mykey.key
                       -out mycert.p12 -name tomcat -CAfile myCA.crt
                       -caname root -chain
```

更复杂的实例，请参考 OpenSSL 文档的相关内容。

下面这个实例展示的是如何利用终端命令行，从零开始创建一个新的 `JKS` 密码存储库，该密码库包含一个自签名的证书。

Windows：

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
```

Unix：

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

（RSA 算法应该作为首选的安全算法，这同样也能保证与其他服务器和组件的大体的兼容性。）

该命令将在用户的主目录下创建一个新文件：`.keystore`。要想指定一个不同的位置或文件名，可以在上述的 `keytool` 命令上添加 `-keystore` 参数，后跟到达 keystore 文件的完整路径名。你还需要把这个新位置指定到 `server.xml` 配置文件上，见后文介绍。例如：

Windows：

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
  -keystore \path\to\my\keystore
```

Unix：

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
  -keystore /path/to/my/keystore
```

执行该命令后，首先会提示你提供 keystore 的密码。Tomcat 默认使用的密码是 `changeit`（全部字母都小写），当然你可以指定一个自定义密码（如果你愿意）。同样，你也需要将这个自定义密码在 `server.xml` 配置文件内进行指定，稍后再予以详述。

接下来会提示关于证书的一般信息，比如组织、联系人名称，等等。当用户试图在你的应用中访问一个安全页面时，该信息会显示给用户，所以一定要确保所提供的信息与用户所期望看到的内容保持一致。

最后，还需要输入密钥密码（key password），这个密码是这一证书（而不是存储在同一密码存储库文件中的其

他证书）的专有密码。`keytool` 提示会告诉你，如果按下回车键，则自动使用密码存储库 keystore 的密码。当然，除了这个密码，你也可以自定义自己的密码。如果选择自定义密码，那么不要忘了在 `server.xml` 配置文件中指定这一密码。

如果操作全部正常，我们现在就会得到一个服务器能使用的有证书的密码存储库文件。

2. 编辑 Tomcat 配置文件

Tomcat 能够使用两种 SSL 实现：

- JSSE 实现，它是Java 运行时（从 1.4 版本起）的一部分。
- APR 实现，默认使用 OpenSSL 引擎。

详细的配置信息依赖于所用的实现方式。如果通过指定通用的 `protocol="HTTP/1.1"` 来配置连接起，那么就会自动选择 Tomcat 所要用到的实现方式。如果安装使用的是 APR（比如你安装了 Tomcat 的原生库），那么它将使用 APR 的 SSL 实现，否则就将使用 Java 所提供的 JSSE 实现。

由于这两种 SSL 实现在 SSL 支持的配置属性上有很大差异，所以强烈建议不要自动选择实现方式。选择实现应该采取这种方式：在连接器的 `protocol` 属性中，通过指定类名来确立实现方式。

定义 Java（JSSE）连接器，不管 APR 库是否已经加载，都可以使用下列方式：

```
<!-- Define a HTTP/1.1 Connector on port 8443, JSSE NIO implementation -->
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" .../>

<!-- Define a HTTP/1.1 Connector on port 8443, JSSE NIO2 implementation -->
<Connector protocol="org.apache.coyote.http11.Http11Nio2Protocol"
           port="8443" .../>

<!-- Define a HTTP/1.1 Connector on port 8443, JSSE BIO implementation -->
<Connector protocol="org.apache.coyote.http11.Http11Protocol"
           port="8443" .../>
```

另一种方法，指定 APR 连接器（APR 库必须可用），则使用：

```
<!-- Define a HTTP/1.1 Connector on port 8443, APR implementation -->
<Connector protocol="org.apache.coyote.http11.Http11AprProtocol"
           port="8443" .../>
```

如果使用 APR，则会出现一个选项，从而可以配置另一种 OpenSSL 引擎。

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="someengine" SSLRandomSeed="somedevice" />
```

默认值为：

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="on" SSLRandomSeed="builtin" />
```

所以要想使用 APR 实现，一定要确保 `SSLEngine` 属性值不能为 `off`。该属性值默认为 `on`，如果指定的是其他值，它也会成为一个有效的引擎名称。

`SSLRandomSeed` 属性指定了一个熵源。生产系统需要可靠的熵源，但熵可能需要大量时间来采集，因此测试系统会使用非阻塞的熵源，比如像“/dev/urandom”，从而能够更快地启动 Tomcat。

最后一步是在 `$CATALINA_BASE/conf/server.xml` 中配置连接器，`$CATALINA_BASE` 表示的是 Tomcat 实例的基本目录。Tomcat 安装的默认 `server.xml` 文件中包含一个用于 SSL 连接器的 `<Connector>` 元素的范例。为了配置使用 JSSE 的 SSL 连接器，你可能需要清除注释并按照如下的方式来编辑它。

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
```

```
<Connector
           protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"/>
```

APR 连接器会使用很多不同的属性来设置 SSL，特别是密钥和证书。 APR 配置范例如下：

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector
           protocol="org.apache.coyote.http11.Http11AprProtocol"
           port="8443" maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           SSLCertificateFile="/usr/local/ssl/server.crt"
           SSLCertificateKeyFile="/usr/local/ssl/server.pem"
           SSLVerifyClient="optional" SSLProtocol="TLSv1+TLSv1.1+TLSv1.2"/>
```

每个属性所用的配置信息与选项都是强制的，可查看 HTTP 连接器配置参考文档中的 SSL 支持部分。一定要确保对所使用的连接器采用正确的属性。BIO、NIO 以及 NIO2 连接器都使用 JSSE，然而APR以及原生的连接器则使用 APR。

`port` 属性指的是 Tomcat 用以侦听安全连接的 TCP/IP 端口号。你可以随意改变它，比如改成 `https` 的默认端口号 443。但是在很多操作系统中，在低于 1024 的端口上运行 Tomcat 必须进行一番特殊的配置，限于篇幅，本文档不再赘述。

如果在这里，你更改了端口号，那么也应该在 非 **SSL**连接器的 **`redirectPort`** 属性值。从而使 **Tomcat** 能够根据 **Servlet** 规范，自动对访问带有安全限制（指定需要 **SSL**）页面的用户进行重定向。

配置完全部信息后，你应该像往常一样，重新启动 Tomcat，从而能够利用 SSL 来访问任何 Tomcat 所支持的 Web 应用了。比如：

```
https://localhost:8443/
```

你将看到跟往常一样的 Tomcat 主页面（除非你修改了 ROOT 应用）。如果出现问题，这样做没有任何效果，请看下面的故障排除小节。

从 CA 处安装证书

如果想从 CA（比如 verisign.com、thawte.com、trustcenter.de）处获取并安装证书，请阅读之前的内容，然后按照下列操作进行：

创建一个本地证书签名请求（CSR）

为了从选择的 CA 处获取证书，必须创建一个证书签名请求（CSR）。CA 通过 CSR 来创建出一个证书，用来证明你的网站是“安全的”。创建 CSR 的步骤如下：

- 创建一个本地自签名证书（如前文所述）：

```
keytool -genkey -alias tomcat -keyalg RSA
-keystore <your_keystore_filename>
```

注意：在某些情况下，为了创建一个能耐生效的证书，你必须在“first- and lastname”字段中输入网站的域名（比如：`www.myside.org`）。

- 然后利用下列代码创建 CSR：

```
keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr
-keystore <your_keystore_filename>
```

现在，我们得到了一个 `certreq.csr` 的文件，可以把它提交给 CA 了（CA 的网站上应有关于如何提交的文档），审核通过后就会收到一个证书。

导入证书

接下来可以把证书导入到本地密钥存储库中。首先你需要把链证书（Chain Certificate）或根证书（Root Certificate）导入到密钥存储库中。然后继续导入证书。

- 从获得证书的 CA 那里下载链证书。 如选择 Verisign.com 的商业证书，则点击：http://www.verisign.com/support/install/intermediate.html。
  如选择 Verisign.com 的试用证书，则点击：http://www.verisign.com/support/verisign-intermediate-ca/Trial_Secure_Server_Root/index.html。
  如选择 Trustcenter.de，则点击：http://www.trustcenter.de/certservices/cacerts/en/en.htm#server。
  如选择 Thawte.com，则点击：http://www.thawte.com/certs/trustmap.html。

- 将链证书导入密钥存储库：

```
keytool -import -alias root -keystore <your_keystore_filename>
-trustcacerts -file <filename_of_the_chain_certificate>
```

- 最后导入你的新证书：

```
keytool -import -alias tomcat -keystore <your_keystore_filename>
-file <your_certificate_filename>
```

疑难排解

以下列出了一些在设置 SSL 通信时经常会遇到的问题及其解决方法：

- 当 Tomcat 启动时，出现这样的异常信息：“ava.io.FileNotFoundException: {some-directory}/{some-file} not found.”

  这很有可能是因为 Tomcat 无法在指定位置处找到密钥存储库。默认情况下，密钥存储库文件应以 `.keystore` 为后缀，位于用户的 home 目录下（也许很你的设置不同）。如果密钥存储库文件在别处，则需要在 Tomcat 配置文件的 `<Factory>` 元素中添加一个 `keystoreFile` 属性。

- 当 Tomcat 启动时，出现这样的异常信息：“java.io.FileNotFoundException: Keystore was tampered with, or password was incorrect.”

  假如排除了有人恶意篡改密钥存储库文件的因素，那么出现这样的异常，最有可能是因为 Tomcat 现在所用的密码不同于你当初创建密钥存储库文件时所用密码。为了解决这个问题，你可以重新创建密钥存储库文件，或者在 Tomcat 配置文件的 `<Connector>` 元素中添加或更新一个 `keystoreFile` 属性。注意：密码都是区分大小写的。

- 当 Tomcat 启动时，出现这样的异常：“java.net.SocketException: SSL handshake error javax.net.ssl.SSLException: No available certificate or key corresponds to the SSL cipher suites which are enabled.”

  出现这样的异常，很有可能是因为 Tomcat 无法在指定的密钥存储库中找到服务器密钥的别名。查看一下 Tomcat 配置文件的 `<Connector>` 元素中所指定的 `keystoreFile` 和 `keyAlias` 值是否正确。注意：`keyAlias` 值可能区分大小写。

如果出现了其他问题，可以求助 **TOMCAT-USER** 邮件列表，你可以在该邮件列表内找到之前消息的归档文件，以及订阅和取消订阅的相关信息。Tomcat 邮件列表的链接是：http://tomcat.apache.org/lists.html。

在应用中使用 SSL 跟踪会话

这是一个 Servlet 3.0 规范中的新功能。由于它将 SSL 会话 ID 关联到物理的客户端服务器连接上，所以导致了一些约束与局限：

- Tomcat 必须有一个属性 **isSecure** 设为 `true` 的连接器。
- 如果 SSL 连接器通过代理或硬件加速器来管理，则它们必须填充 SSL 请求报头（参见 SSLValve），从而使 SSL 会话 ID 可见于 Tomcat。
- 如果 Tomcat 终止了 SSL 连接，则不可能采用会话重复，因为 SSL 会话 ID 在两个端点处都是不一样的。

为了开启 SSL 会话跟踪，只需使用上下文侦听器将上下文的跟踪模式设定为 SSL 即可（如果还开启了其他跟踪模式，则会优先使用它）。如下所示：

```
package org.apache.tomcat.example;

import java.util.EnumSet;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.SessionTrackingMode;

public class SessionTrackingModeListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        // Do nothing
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext context = event.getServletContext();
        EnumSet<SessionTrackingMode> modes =
            EnumSet.of(SessionTrackingMode.SSL);

        context.setSessionTrackingModes(modes);
    }

}
```

注意：SSL 会话跟踪是针对 BIO、NIO 以及 NIO2 连接器来实现的，目前还没有针对 APR 连接器的实现。

其他技巧

要想从请求中访问 SSL 会话 ID，可使用：

```
String sslID =
(String)request.getAttribute("javax.servlet.request.ssl_session_id");
```

关于这一方面的其他内容，可参看Bugzilla。

为了终止 SSL 会话，可以使用：

```
// Standard HTTP session invalidation
session.invalidate();

// Invalidate the SSL Session
org.apache.tomcat.util.net.SSLSessionManager mgr =
    (org.apache.tomcat.util.net.SSLSessionManager)
    request.getAttribute("javax.servlet.request.ssl_session_mgr");
mgr.invalidateSession();

// Close the connection since the SSL session will be active until the connection
// is closed
response.setHeader("Connection", "close");
```

注意：由于使用了 SSLSessionManager 类，所以这段代码是专门针对 Tomcat 的。当前只适用于 BIO、NIO 以及 NIO2 连接器，不适合 APR/原生连接器。

13

SSI（服务器端嵌入）

简介

SSI（服务器端嵌入）是一组放在 HTML 页面中的指令，当服务器向客户端访问提供这些页面时，会解释执行这些指令。它们能为已有的 HTML 页面添加动态生成内容，不需要通过 CGI 程序来或其他的动态技术来重新改变整个页面。

如果利用 Tomcat 作为 HTTP 服务器并需要 SSI 支持时，可以添加 SSI 支持。通常，如果你运行的不是像 Apache 那样的服务器，就通过开发来实现这种支持。

Tomcat SSI 支持实现了与 Apache 完全一致的 SSI 指令。关于使用 SSI 指令的详细信息，可参考Apache 的 SSI 简介。

SSI 支持可以有两种方式来实现：servlet 或过滤器。你只能利用其中的一种方式来提供 SSI 支持。

基于 servlet 的 SSI 支持是通过 `org.apache.catalina.ssi.SSIServlet` 类来实现的。一般来说，这个 servlet 映射至 URL 模式"*.shtml"。

基于过滤器的 SSI 支持则利用 `org.apache.catalina.ssi.SSIFilter` 类来实现。一般而言，该过滤器映射至 URL 模式 "*.shtml"，但是它也可以被映射至 "*"，因为它会基于 MIME 类型选择性地启用/禁用对 SSI 的处理。初始参数 `contentType` 允许你将 SSI 处理应用于 JSP 页面、JavaScript 内容以及其他内容中。

默认 Tomcat 是不支持 SSI 的。

安装

警告：SSI 指令可用于执行 Tomcat JVM 之外的程序。如果使用 Java SecurityManager，它会绕过你在 `catalina.policy` 中配置的安全策略。

为了使用 SSI servlet，要从 `$CATALINA_BASE/conf/web.xml` 中去除 SSI servlet 及 servlet 映射配置旁边的 XML 注释。

为了使用 SSI 过滤器，要从 `$CATALINA_BASE/conf/web.xml` 中去除 SSI 过滤器及过滤器映射配置旁边的 XML 注释。

只有标明为 privileged 的上下文才可以使用 SSI 功能（参看 Context 元素的 privileged 属性）。

Servlet 配置

以下这些 servlet 初始化参数可以配置 SSI servlet 的行为：

- **buffered** 是否应缓存该 servlet 的输出？（0 = false，1 = true）默认为 0（false）。
- **debug** 调试 servlet 所记录信息的调试细节度。默认为 0。
- **expires** 包含 SSI 指令的页面失效的秒数。默认行为针对的是每个请求所应执行的所有 SSI 指令。
- **isVirtualWebappRelative** “虚拟”的 SSI 指令路径是否应被解释为相对于上下文根目录的相对路径（而不是服务器根目录）？默认为 false。
- **inputEncoding** 如果无法从资源本身确定编码，则应指定给 SSI 资源的编码。默认为系统默认编码。
- **outputEncoding** 用于 SSI 处理结果的编码。默认为 UTF-8。
- **allowExec** 是否启用 `exec` 命令？默认为 false。

过滤器配置

以下这些过滤器初始化参数可以配置 SSI 过滤器的行为：

- **contentType** 在应用 SSI 处理之前必须匹配的正则表达式模式。在设计自己的模式时，不要忘记 MIME 内容类型后面可能会带着可选的字符集：“mime/type; charset=set”。默认为 “text/x-server-parsed-html(;.*)?”。
- **debug** 调试 servlet 所记录信息的调试细节度。默认为 0。
- **expires** 包含 SSI 指令的页面失效的秒数。默认行为针对的是每个请求所应执行的所有 SSI 指令。
- **isVirtualWebappRelative** “虚拟”的 SSI 指令路径是否应被解释为相对于上下文根目录的相对路径（而不是服务器根目录）？默认为 false。
- **allowExec** 是否启用 `exec` 命令？默认为 false。

指令

指令采取 HTML 注释的形式。在将页面发送到客户端之前，解读指令，并用所得结果来替换指令。指令的一般形式为：

```
<!--#directive [parm=value] -->
```

这些指令包括：

- **config** `<!--#config timefmt="%B %Y" -->` 用于设定日期格式以及其他一些 SSI 处理的项目。
- **echo** `<!--#echo var="VARIABLE_NAME" -->` 将被变量值所取代。
- **exec** 在主机系统上用于运行命令。
- **include** `<!--#include virtual="file-name" -->` 插入内容。
- **flastmod** `<!--#flastmod file="filename.shtml" -->` 返回文件最后修改的时间。
- **fsize** `<!--#fsize file="filename.shtml" -->` 返回文件大小。
- **printenv** `<!--#printenv -->` 返回所有定义变量的列表。
- **set** `<!--#set var="foo" value="Bar" -->` 为用户自定义变量赋值。
- **if elif endif else** 用于生成条件部分，例如：

```
<!--#config timefmt="%A" -->
<!--#if expr="$DATE_LOCAL = /Monday/" -->
<p>Meeting at 10:00 on Mondays</p>
<!--#elif expr="$DATE_LOCAL = /Friday/" -->
<p>Turn in your time card</p>
<!--#else -->
<p>Yoga class at noon.</p>
<!--#endif -->
```

关于使用 SSI 指令的详细信息，可参考Apache 的 SSI 简介。

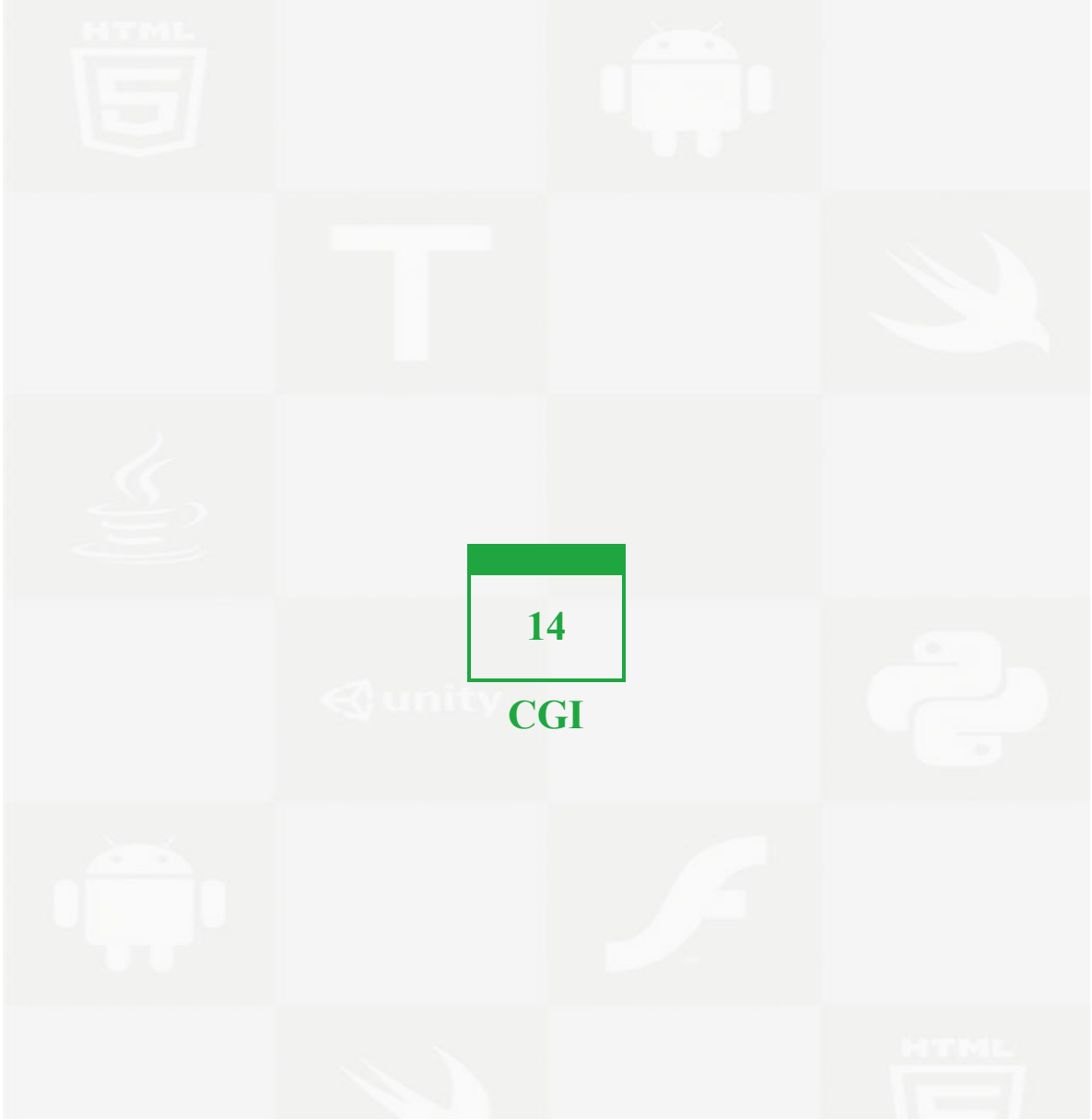
变量

SSI servlet 当前能实现下列变量：

变量名	描述
AUTH_TYPE	用于这些用户的验证类型：BASIC、FORM，等等。
CONTENT_LENGTH	从表单传入数据的长度（以字节或字符数）
CONTENT_TYPE	查询数据的 MIME 类型。比如：text/html
DATE_GMT	以格林威治标准时间（GMT）表示的当前时间与日期
DATE_LOCAL	以本地时区表示的当前日期与时间
DOCUMENT_NAME	当前文件名
DOCUMENT_URI	文件的虚拟路径
GATEWAY_INTERFACE	服务器所使用的通用网关接口（CGI）的修订版本，比如：CGI/1.1
HTTP_ACCEPT	客户端能够接受的 MIME 类型列表
HTTP_ACCEPT_ENCODING	客户端能够接受的压缩类型列表
HTTP_ACCEPT_LANGUAGE	客户端能够接受的语言类型列表
HTTP_CONNECTION	如何管理与客户端的连接："Close" 或 "Keep-Alive"
HTTP_HOST	客户端所请求的网站

HTTP_REFERER	客户端链接的文档的 URL
HTTP_USER_AGENT	客户端用于处理请求的浏览器
LAST_MODIFIED	当前文档的最后修改日期与时间
PATH_INFO	传入 servlet 的额外路径信息
PATH_TRANSLATED	变量 `PATH_INFO` 所提供路径的转换版本
QUERY_STRING	在 URL 中，跟在 ？ 后面的查询字符串
QUERY_STRING_UNESCAPED	带有所有经过 `\` 转义的shell 元字符的未解码查询字符串
REMOTE_ADDR	用户作出请求的远端 IP 地址
REMOTE_HOST	用户作出请求的远端主机名
REMOTE_PORT	用户作出请求的远端 IP 地址的端口号
REMOTE_USER	用户的认证名称
REQUEST_METHOD	处理信息请求的方法：`GET`、`POST`，等等
REQUEST_URI	客户端所请求的最初页面
SCRIPT_FILENAME	当前页面在服务器上的位置
SCRIPT_NAME	页面名称
SERVER_ADDR	服务器的 IP 地址
SERVER_NAME	服务器的主机名或 IP 地址
SERVER_PORT	服务器接收请求的端口

SERVER_PROTOCOL	服务器所使用的协议，比如：HTTP/1.1
SERVER_SOFTWARE	应答客户端请求的服务器软件的名称与版本号
UNIQUE_ID	用于识别当前会话（如果存在）的令牌



14

CGI

简介

CGI（通用网关接口）定义了一种 Web 服务器与外部内容生成程序的交互方式，这里所说的外部内容生成程序通常被称为 CGI 程序或 CGI 脚本。

当你使用 Tomcat 做为 HTTP 服务器，并且需要 CGI 支持时，可以在 Tomcat 中添加 CGI 支持。Tomcat 的 CGI 支持很大程度上能够跟 Apache 的httpd's 相兼容，但也存在一些局限（比如只有一个 cgi-bin 目录）。

CGI 支持是通过 servlet 类 `org.apache.catalina.servlets.CGIServlet` 来实现的。一般而言，该 servlet 与 URL 模式“/cgi-bin/*” 相对应。

Tomcat 默认不支持 CGI。

安装

> 警告：CGI 脚本用于执行 Tomcat JVM 外部的程序。如果使用 Java 的 SecurityManager，则它将绕过 `catalina.policy` 中配置的安全策略。

为了启用 CGI 支持：

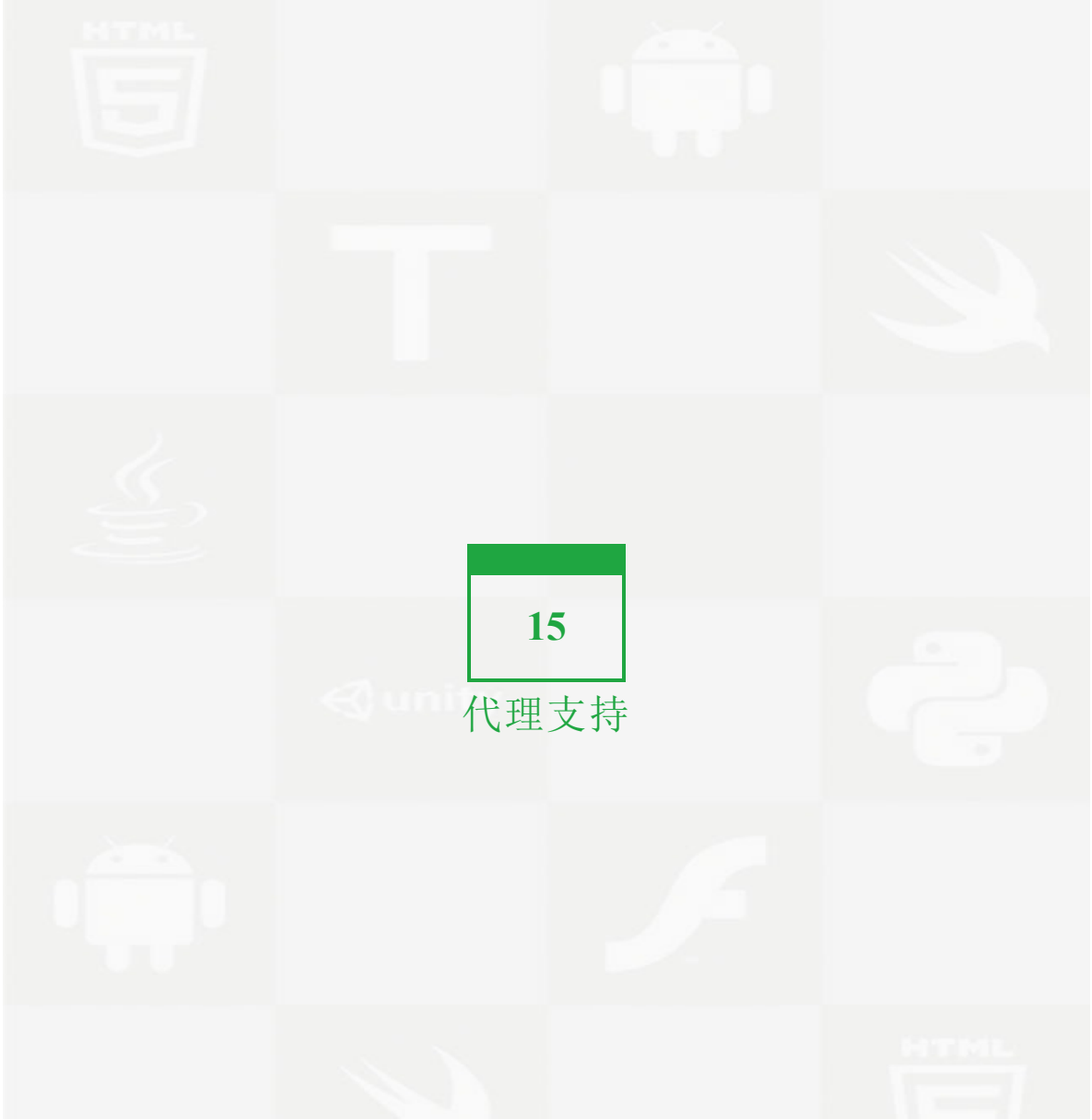
1. 在默认的 `$CATALINA_BASE/conf/web.xml` 文件中，存在被注释掉的用于 CGI servlet 的范例 servlet 及 servlet-mapping 元素。在 Web 应用中启用 CGI 支持，需要将 servlet 和 servlet-mapping 声明都复制到 Web 应用的 `WEB-INF/web.xml` 文件中。
2. 在 Web 应用中的 Context 元素中设置 `privileged="true"`。

   只有享有特权的上下文才能被允许使用 CGI servlet。注意，修改全局 `$CATALINA_BASE/conf/context.xml` 文件会影响所有的 Web 应用。查阅 Context 文档来了解详情。

配置

下面是用来配置 CGI servlet 行为的一些 Servlet 初始参数：

- **cgiPathPrefix** 搜索 CGI 脚本的路径，一般从 Web 应用根目录 + 文件.分隔符 + 这个前缀开始搜索。该参数默认为空值，从而使得 Web 应用根目录被用作搜索路径。建议取值为：`WEB-INF/cgi`。
- **debug** 该 servlet 所记录信息调试细节度。默认为 `0`。
- **executable** 用于运行脚本的可执行文件后缀名，如果脚本自身就是可执行文件（比如 .exe 文件），则可以将该参数显式设置为空的字符串。默认是 `perl`，即默认是 perl 脚本。
- **executable-arg-1** 与 **executable-arg-2**，等等 `executable` 的其他参数。它们位于 CGI 脚本名称之前。默认不存在其他额外参数。
- **parameterEncoding** CGI Servlet 所使用的参数编码名称，默认为 `System.getProperty("file.encoding","UTF-8")`。首选系统默认编码，如果系统属性不可用，则采用 UTF-8 编码。
- **passShellEnvironment** 是否应将 Tomcat 过程的 shell 环境变量（如果存在）传入 CGI 脚本？默认为 `false`。
- **stderrTimeout** 在终止 CGI 过程之前，等待标准错误输出信息（stderr）读取完毕的时间（以毫秒计）。默认为 `2000`。



15

代理支持

简介

使用 Tomcat 的标准配置，Web 应用可以请求服务器名称和端口号》。当 Tomcat 单独和 HTTP/1.1 连接器运行时，通常会报告指定在请求中的服务器名称，以及连接器正在侦听的端口号。servlet API：

- `ServletRequest.getServerName()` 返回接收请求的服务器主机名。
- `ServletRequest.getServerPort()` 返回接收请求的服务器端口号。
- `ServletRequest.getLocalName()` 返回接收请求的 IP 接口的主机名。
- `ServletRequest.getLocalPort()` 返回接收请求的 IP 接口的端口号。

当你在代理服务器后（或者配置成具有代理服务器特征行为的 Web 服务器）运行时，可能有时会更愿意管理通过这些调用产生的值。特别是，你一般会希望端口号反应指定在原始请求中的值，而非连接器所正在侦听的那个值。可以使用 `<Connector>` 元素中的 `proxyName` 和 `proxyPort` 属性来配置这些值。

代理支持可以采取的形式有很多种。下面来讨论适用于一些通常情况的代理配置。

Apache 1.3 代理支持

Apache 1.3 支持一种可选模式（`mod_proxy`），可以将 Web 服务器配置成代理服务器，从而将对于特定 Web 应用的请求转发给 Tomcat 实例，不需要配置 Web 连接器（比如说 `mod_jk`）。为了达成这一目标，需要执行下列任务：

1.配置 Apache，使其包含 `mod proxy` 模块。如果是从源码开始构建，最简单的方式是在 `./configure` 命令行中包括 `--enable-module=proxy` 指令。

2.如果没有添加 `mod_proxy` 模块，则检查一下是否在 Apache 启动时加载了该模块，在 `httpd.conf` 文件中使用下指令：

```
LoadModule proxy_module  {path-to-modules}/mod_proxy.so
AddModule  mod_proxy.c
```

3.在 `httpd.conf` 文件中包括两个指令。分别为两个要转交给 Tomcat 的 Web 应用。例如，转交上下文路径 `/myapp` 处的应用，则需要如下指令：

```
ProxyPass          /myapp  http://localhost:8081/myapp
ProxyPassReverse   /myapp  http://localhost:8081/myapp
```

上述指令告诉 Apache 将 `http://localhost/myapp/*` 形式的 URL 转交给在端口 8081 侦听的 Tomcat 连接器。

4.配置 Tomcat，使其包含一个特殊的 `<Connector>` 元素，并配置好相应的代理设置。范例如下所示：

```
<Connector port="8081" ...
              proxyName="www.mycompany.com"
              proxyPort="80"/>
```

这将导致该 Web 应用内的 servlet 认为，所有代理请求都指向的是 80 端口处的 `www.mycompany.com`。

5.可以忽略 `<Connector>` 元素的 `proxyname` 属性，这是完全合法的。如果忽略，那么 `request.getServerName()` 返回值将是运行 Tomcat 的主机名——对于该例而言，它就是 `localhost`。

6.如果有一个 `<Connector>`（内嵌于同一 Service 元素之中）在 8080 端口处侦听。则针对这两个端口之中任何一个端口的请求将共享同样的虚拟主机和 Web 应用。

7.你可以利用所在操作系统的 IP 过滤功能来限制与 8081 端口的连接。（在该例中），使其跟 8081 端口的连接只能从运行 Apache 的服务器上

8.或者可以采用另外一种方式：可以设置一系列只能通过代理访问的 Web 应用，如下所示：

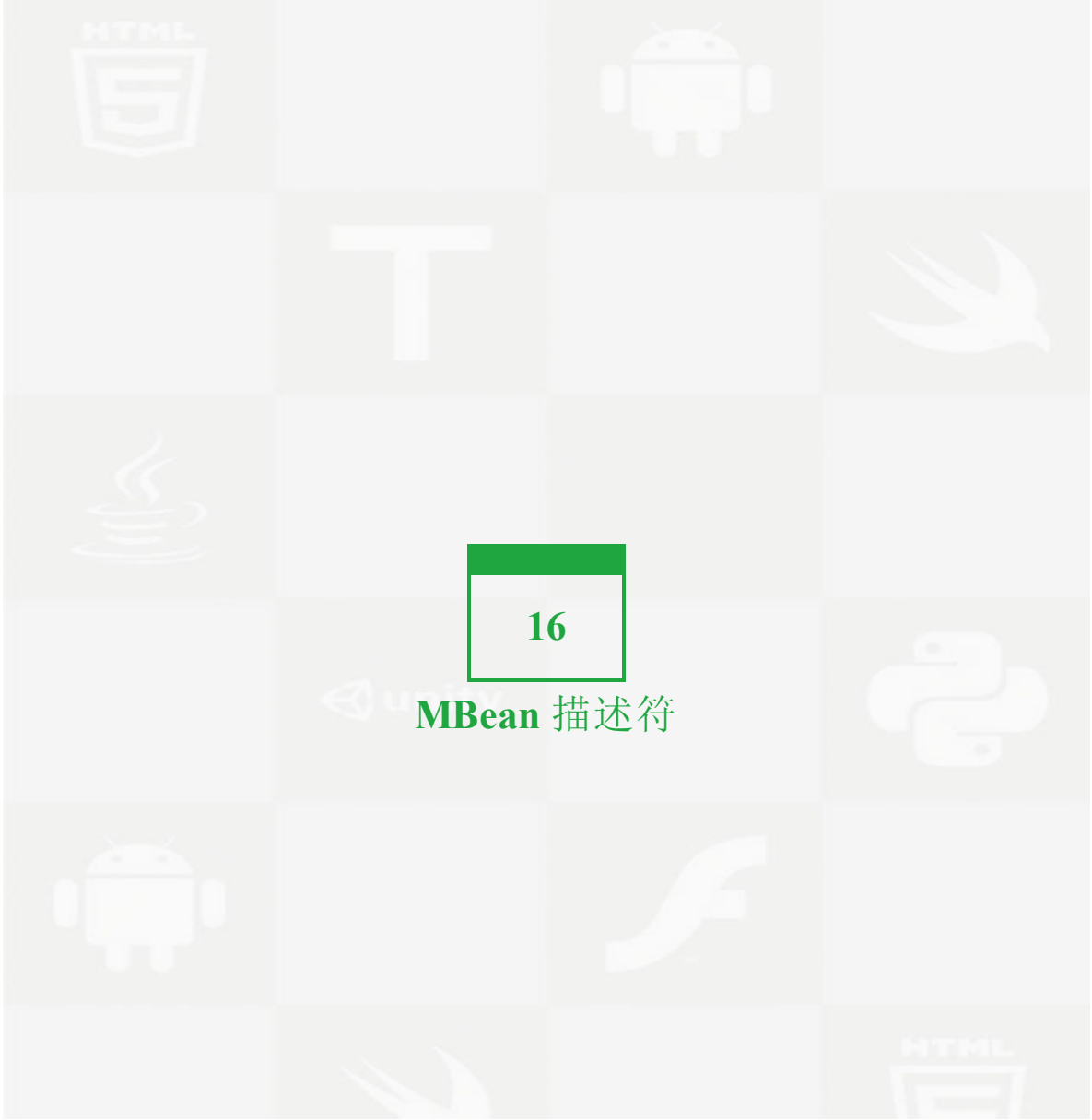
- 为代理端口配置另一个只包含一个 `<Connector>` 的 `<Service>`。
- 为能通过代理访问的虚拟主机和 Web 应用配置适宜的 Engine、Host，以及 Context 元素。
- 另外，还可以选择利用 IP 过滤器保护端口 8081，如前文所述。

9.当请求被 Apache 所代理处理时，Web 服务器会在访问日志中记下这些请求，所以通常应该禁止 Tomcat 本身执行访问记录。

通过以上介绍的这种方式来代理请求，所有针对已经配置过的 Web 应用的请求（包括针对静态内容的请求）都将由 Tomcat 处理。你可以通过 Web 连接器 `mod_jk`（而不是 `mod_proxy`）来提高性能。通过配置 `mod_jk`，还可以让 Web 服务器提供静态内容服务，这些静态内容没有受到过滤器的处理，或者被 Web 应用部署描述符文件中所定义的安全限制所束缚。

Apache 2.0 代理支持

和 Apache 1.3 中的指令大致相同，只不过在 Apache 2.0 中，可以省略 `AddModule mod_proxy.c`。



16

MBean 描述符

简介

Tomcat 使用 JMX MBean 来实现自身的性能管理。

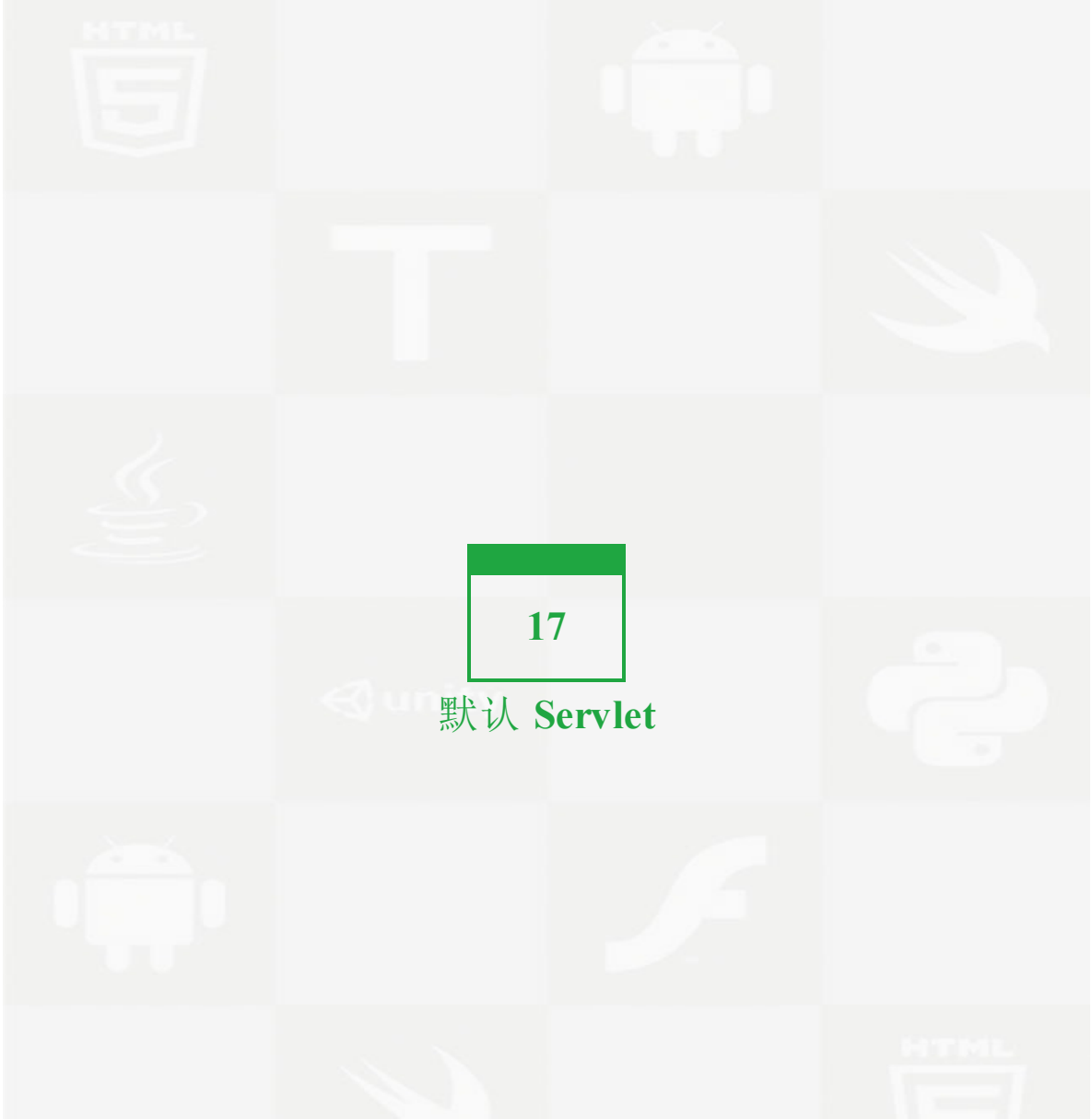
每个包里的 mbeans-descriptor.xml 是针对 Catalina 的 JMX MBean 描述。

为了避免出现 “ManagedBean is not found” 异常，你需要为自定义组件添加 MBean 描述。

添加 **Mbean** 描述

在 mbeans-descriptor.xml 文件中，你可以为自定义组件添加 Mbean 描述。这个 xml 文件跟它所描述的类文件同在一个包内。

```
<mbean          name="LDAPRealm"
          className="org.apache.catalina.mbeans.ClassNameMBean"
        description="Custom LDAPRealm"
             domain="Catalina"
              group="Realm"
               type="com.myfirm.mypackage.LDAPRealm">

  <attribute   name="className"
        description="Fully qualified class name of the managed object"
               type="java.lang.String"
          writeable="false"/>

  <attribute   name="debug"
        description="The debugging detail level for this component"
               type="int"/>
  .
  .
  .

</mbean>
```



17

默认 Servlet

什么是 **DefaultSevelet**

DefaultSevelet 是处理静态资源的 Sevelet。

在什么位置声明它？

它在 `$CATALINA_HOME/conf/web.xml` 中被全局声明。默认形式的声明是这样的：
`$CATALINA_HOME/conf/web.xml`

```
    <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>
          org.apache.catalina.servlets.DefaultServlet
        </servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>0</param-value>
        </init-param>
        <init-param>
            <param-name>listings</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

...

    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

因此在默认的情况下，默认 servlet 在 Web 应用启动时被装载，目录列表可被使用，日志调试功能被关掉。

What can I Change?

DefaultServlet 允许设置以下初始化参数：

属性	描述
`debug`	调试级别。如果不是 tomcat 开发人员，则没有什么太大的用处。截止本文写作时，有用的值是 0、1、11、1000。默认值为 0
`listings`	如果没有欢迎文件，要不要显示目录列表？值可以是true 或 false。 欢迎文件是servlet api的一部分。 警告：目录列表中含有的很多项目都是非常消耗服务性能的，如果对大型目录列表多次进行请求，会严重消耗服务器资源。
`gzip`	如果某个文件存在gzip格式的文件（带有`gz`后缀名的文件通常就在原始文件旁边）。如果用户代理支持 gzip 格式，并且启用了该选项，Tomcat 就会提供该格式文件的服务。默认为 false。 如果直接请求带有 `gz` 后缀名的文件，是可以访问它们的，所以如果原始资源受安全挟制的保护，则 gzip 文件也同样是受保护的。
`readmeFile`	如果提供了目录列表，那么可能也会提供随带的 readme 文件。这个文件是被插入的，因此可能会包含 HTML。默认值是 null。
`globalXsltFile`	如果你希望自定义目录列表，你可以使用一个 XSL 转换（transformation）。这个值是一个可用于所有目录列表的相对路径文件名（既相对于 `CATALINA_BASE/conf/` 也相对于 `$CATALINA_HOME/conf/`）。》这可以每个上下文或每一目录》可参看下面介

	绍的 `contextXsltFile` 和 `localXsltFile`。该 xml 文件的格式会在下文介绍。
`contextXsltFile`	你可以通过 `contextXsltFile` 来自定义你的目录列表。这必须是一个上下文相对路径（例如：`/path/to/context.xslt`），相对于带有 `.xsl` 或 `.xslt` 扩展名的文件。它将覆盖 `globalXsltFile`。如果提供了该值，但相对文件却不存在，则将使用 `globalXsltFile`。如果 `globalXsltFile` 也不存在，则显示默认的目录列表。
`localXsltFile`	你还可以在每个目录通过配置 `localXsltFile` 定制你的目录列表。它应该是在产生列表的目录里的一个相对路径文件名。它覆盖 `globalXsltFile` 和 `contextXsltFile`。如果该值存在，但是文件不存在，那么就使用 `contextXsltFile`。如果 `contextXsltFile` 也不存在，那么就会使用 `globalXsltFile`。如果 `globalXsltFile` 也不存在，那么默认的目录列表就会被显示出来。
`input`	在读取用于服务的资源时的输入缓冲大小（以字节计）。默认为 2048。
`output`	在修改用于服务的资源时的输出缓冲大小（以字节计）。默认为 2048。
`readonly`	上下文是否为“只读”，从而拒绝执行 PUT 或 DELETE 这样的 HTTP 命令。默认为 true。
`fileEncoding`	文件编码用于读取静态资源时。默认取平台默认值。

`sendfileSize`	如果所用的连接器支持 sendfile，这个参数表示所用的 sendfile 最小的文件大小（以 KB 计）。使用负数往往表示可以禁止使用 sendfile。默认为 48。
`useAcceptRanges`	如果为 true，则将设置 Accept-Ranges 报头，在适于响应时。
`showServerInfo`	当使用目录列表，服务器信息是否应该提供给发往客户端的响应中。默认为 true。

我该如何自定义目录列表

你可以用自定义实现来覆盖 DefaultServlet，并将它用在 web.xml 声明中。如果你能明白刚才所说的是什么意思，我们就认为你能读懂 DefaultServlet 的代码并作出适当的调整。（如果不能明白，则说明这种方式不适合你。）

`localXsltFile` 或 `globalXsltFile`

格式如下：

```
    <listing>
     <entries>
      <entry type='file|dir' urlPath='aPath' size='###' date='gmt date'>
        fileName1
      </entry>
      <entry type='file|dir' urlPath='aPath' size='###' date='gmt date'>
        fileName2
      </entry>
      ...
     </entries>
     <readme></readme>
    </listing>
```

- 如果 `type = 'dir'`，则 size 丢失。
- Readme 是一个 CDATA 项。

下面是一个能够模仿 Tomcat 默认行为的范例 xsl 文件：

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0">

  <xsl:output method="html" html-version="5.0"
    encoding="UTF-8" indent="no"
    doctype-system="about:legacy-compat"/>

  <xsl:template match="listing">
   <html>
    <head>
      <title>
        Sample Directory Listing For
        <xsl:value-of select="@directory"/>
      </title>
      <style>
        h1 {color : white;background-color : #0086b2;}
        h3 {color : white;background-color : #0086b2;}
        body {font-family : sans-serif,Arial,Tahoma;
              color : black;background-color : white;}
        b {color : white;background-color : #0086b2;}
        a {color : black;} HR{color : #0086b2;}
        table td { padding: 5px; }
      </style>
    </head>
    <body>
      <h1>Sample Directory Listing For
            <xsl:value-of select="@directory"/>
      </h1>
      <hr style="height: 1px;" />
      <table style="width: 100%;">
```

```
        <tr>
          <th style="text-align: left;">Filename</th>
          <th style="text-align: center;">Size</th>
          <th style="text-align: right;">Last Modified</th>
        </tr>
        <xsl:apply-templates select="entries"/>
        </table>
      <xsl:apply-templates select="readme"/>
      <hr style="height: 1px;" />
      <h3>Apache Tomcat/<version-major-minor/></h3>
    </body>
   </html>
  </xsl:template>


  <xsl:template match="entries">
    <xsl:apply-templates select="entry"/>
  </xsl:template>

  <xsl:template match="readme">
    <hr style="height: 1px;" />
    <pre><xsl:apply-templates/></pre>
  </xsl:template>

  <xsl:template match="entry">
    <tr>
      <td style="text-align: left;">
        <xsl:variable name="urlPath" select="@urlPath"/>
        <a href="{$urlPath}">
          <pre><xsl:apply-templates/></pre>
        </a>
      </td>
      <td style="text-align: right;">
        <pre><xsl:value-of select="@size"/></pre>
      </td>
      <td style="text-align: right;">
        <pre><xsl:value-of select="@date"/></pre>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

如何保证目录列表的安全性

在每一个单独的 Web 应用中使用 web.xml。可查看 Servlet 规范的安全性部分的内容。

18

集群化与会话复制

重要说明

相关内容详情可以查看集群配置文档

只需将下列信息放入 `<Engine>` 或 `<Host>` 元素即可实现集群：

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/>
```

上述配置启用了全局（all-to-all）会话复制功能，全局会话复制是指利用 `DeltaManager` 来只复制会话中的变更（*Session Delta*，也译作“会话增量”）。这里说的“全局”是指：会话变更会被复制到集群中的所有其他节点（指 Tomcat 实例）中。全局复制非常适于小集群，但不建议在大集群（包含很多 Tomcat 节点）上采用这种方法。另外，值得注意的是，当使用 `delta manager` 时，它会将变更复制到所有的节点上，甚至包括那些根本没有部署该应用的节点。

为了解决这个问题，你就得使用 **BackupManager**。它会把会话数据复制给一个指定的备份节点（这种复制也被称为“配对复制”），而且该备份节点也一定要部署了相关应用。BackupManager 的缺点在于：不像 DeltaManager 那样久经实践考验。

下面是一些重要的默认值。

1. IP 组播地址为：228.0.0.4
2. IP 组播端口为：45564（端口和地址一起确定了集群成员）。
3. 广播的 IP 是 `java.net.InetAddress.getLocalHost().getHostAddress()`（你一定不能广播 127.0.0.1，这是一个常见错误。）
4. 侦听复制信息的 TCP 端口是在 4000 - 4100 之间遇到的第一个能用的服务器套接字。
5. 两个侦听器都配置有 `ClusterSessionListener`。
6. 两个拦截器都配置有 `TcpFailureDetector` 和 `MessageDispatch15Interceptor`。

下面是默认的集群配置：

```
        <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
                 channelSendOptions="8">

          <Manager className="org.apache.catalina.ha.session.DeltaManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"/>

          <Channel className="org.apache.catalina.tribes.group.GroupChannel">
            <Membership
className="org.apache.catalina.tribes.membership.McastService"
                        address="228.0.0.4"
                        port="45564"
                        frequency="500"
                        dropTime="3000"/>
            <Receiver
className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                      address="auto"
                      port="4000"
                      autoBind="100"
                      selectorTimeout="5000"
                      maxThreads="6"/>

            <Sender
className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
              <Transport
className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
            </Sender>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
```

```
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Intercept
or"/>
          </Channel>

          <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
                 filter=""/>
          <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>

          <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                    tempDir="/tmp/war-temp/"
                    deployDir="/tmp/war-deploy/"
                    watchDir="/tmp/war-listen/"
                    watchEnabled="false"/>

          <ClusterListener
className="org.apache.catalina.ha.session.ClusterSessionListener"/>
        </Cluster>
```

稍后，本文档将更详细地阐述这部分的内容。

集群基本知识

要想在 Tomcat 8 上运行会话复制，需要执行以下步骤：

- 所有的会话属性必须实现 `java.io.Serializable`。
- 在 server.xml 中取消注释 `Cluster` 元素。
- 如果你已经定义了自定义集群值，确保在 server.xml 中的 Cluster 元素下面也定义了 `ReplicationValve`。
- 如果你的多个 Tomcat 实例都运行在同一台机器上，则要确保每个实例都具有唯一的 `tcpListenPort`。通常 Tomcat 会自行解决这个问题，会在 4000 - 4100 上自动侦测可用的端口。
- 确保 `web.xml` 含有 `<distributable/>` 属性。
- 如果使用 mod_jk，则要确保在 `<Engine name="Catalina" jvmRoute="node01" >` 上设定 `jvmRoute` 属性。`jvmRoute` 属性值必须匹配 workers.properties 中的 worker 名称。
- 所有的节点必须具有相同的时间，并且与 NTP 服务器同步。
- 确保负载均衡配置了会话模式.

负载均衡可以通过多种技术来实现，参看负载均衡部分。

注意：会话状态是通过 cookie 来记录的，所以你的 URL 必须保持一致，否则就会创建一个新会话。

注意：当前如要支持集群，需要 JDK 1.5 或更新版本。

集群模块使用 Tomcat 的 JULI 日志框架，所以可以通过 logging.properties 文件来配置日志。为了跟踪消息，你可以启用 `org.apache.catalina.tribes.MESSAGES` 键上的日志。

概述

在 Tomcat 中，可以使用以下方法中的一种启用会话复制：

1. 使用会话持久性，将会话保存到共享文件系统中（PersistenceManager + FileStore）。
2. 使用会话持久性，将会话保存到共享数据库中（PersistenceManager + JDBCStore）。
3. 使用内存复制，使用 Tomcat 自带的 SimpleTcpCluster（lib/catalina-tribes.jar + lib/catalina-ha.jar）。

在这一版本的 Tomcat 中，可以使用 `DeltaManager` 执行全局式会话状态复制，或者使用 `BackupManager` 执行备份复制，将会话复制到一个节点上。全局式会话复制这种算法只有在集群较小时才比较有效。对于大型集群，更多使用主从会话复制，将会话存储到一台配置了 BackupManager 的备份服务器上。

当前可以使用域名 worker 属性（mod_jk 版本 > 1.2.8）来构建集群分区，从而有可能利用 DeltaManager 实现更具有可扩展性的集群方案（需要为此配置域的拦截器）。为了在全局性环境中降低网络流量，可以将集群分成几个较小的分组。为不同的分组使用不同的组播地址即能实现这种方案。下图展示的是一种简单的配置方案。

```
DNS 轮询
|
负 载 均 衡 器
/               \
集群 1              集群 2
/     \             /     \
Tomcat 1 Tomcat 2  Tomcat 3 Tomcat 4
```

值得注意的是，使用会话复制仅仅是集群化的一个基础方案。关于集群的实现，另一个常用的概念是耕种（farming），比如：只需将应用部署到一个服务器上，集群就会将部署分发到整个集群的各个节点中。这都是 FarmWarDeployer 所具有的功能（参看 `server.xml` 中的集群范例）。

下一节将深入介绍会话复制的工作原理以及配置方式。

集群信息

通过组播心跳包（heartbeat）建立起成员（Membership）关系，因此，如果希望细分集群，可以改变 `<Membership>` 元素中的组播 IP 地址或端口。

心跳包中含有 Tomcat 节点的 IP 地址，以及 Tomcat 用来侦听会话复制流量的 TCP 端口。所有的数据通信都使用了 TCP 协议。

`ReplicationValve` 用于查找请求结束的时间，如果存在会话复制，就对该复制进行初始化。只复制会话变更的数据（通过在会话上调用 `setAttribute` 或 `removeAttribute` 来完成）。

复制的异步与同步模式应该是最值得我们注意的一个特点了。在同步复制模式下，复制的会话通过线缆传送，重新在所有集群节点上实例化，这样才会返回请求。同步和异步是通过 `channelSendOptions` 标志（整型值）来配置的。`SimpleTcpCluster/DeltaManager` 组合的默认值是 8，从而是异步。详情可以参考一下 send flag(overview) 或 send flag(javadoc)。在异步复制过程中，请求不必等到数据被复制完毕即可返回。异步复制缩短了请求时间，而同步复制则保证了能在请求返回之前复制完会话。

当发生崩溃时，将会话绑定到故障转移节点

如果你使用了 mod_jk 而没有使用粘性会话（sticky session），或者粘性会话由于某种原因而不起作用，或者仅是故障转移，会话 id 需要修改，因为它之前含有之前 Tomcat 的 worker id（通过 Engine 元素中的 jvmRoute 定义）。为了解决这个问题，就要用到 JvmRouteBinderValve。

JvmRouteBinderValve 将重写会话 id，以便确保下一个请求在故障转移后依然能保持粘性（不会因为 worker 不再可用而回滚到某个随机的节点中）。利用同样的名字，该值重写了 cookie 中的 JSESSIONID 值。假如没有正确地设置 valve，将使 mod_jk 模块在失败后很难保持会话的粘性。

记住，如果在 server.xml 中自定义值，那么默认值将不再有效，所以一定要确保添加了默认所定义的值。

提示：

利用属性 **sessionIdAttribute** 可以改变包含旧会话 id 的请求属性名。默认的请求属性名是：*org.apache.catalina.ha.session.JvmRouteOrignalSessionID*。

技巧：

可以启用 mod_jk 翻转模式在删除一个节点， 然后启用了 mod_jk Worker 禁用 JvmRouteBinderValves 。这种用例意味着只有请求的会话才能得到迁移。

配置范例

```
        <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
                 channelSendOptions="6">

          <Manager className="org.apache.catalina.ha.session.BackupManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"
                   mapSendOptions="6"/>
          <!--
          <Manager className="org.apache.catalina.ha.session.DeltaManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"/>
          -->
          <Channel className="org.apache.catalina.tribes.group.GroupChannel">
            <Membership
className="org.apache.catalina.tribes.membership.McastService"
                          address="228.0.0.4"
                          port="45564"
                          frequency="500"
                          dropTime="3000"/>
            <Receiver
className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                        address="auto"
                        port="5000"
                        selectorTimeout="100"
                        maxThreads="6"/>

            <Sender
className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
              <Transport
className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
            </Sender>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Intercept
or"/>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.ThroughputInterceptor"/>
          </Channel>

          <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"

filter=".*\.gif|.*\.js|.*\.jpeg|.*\.jpg|.*\.png|.*\.htm|.*\.html|.*\.css|.*\.txt"/>

          <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                    tempDir="/tmp/war-temp/"
                    deployDir="/tmp/war-deploy/"
                    watchDir="/tmp/war-listen/"
                    watchEnabled="false"/>

          <ClusterListener
className="org.apache.catalina.ha.session.ClusterSessionListener"/>
        </Cluster>
```

下面来仔细剖析一下这段代码。

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
```

```
channelSendOptions="6">
```

Cluster 是主要元素，可在该元素内配置所有的集群相关细节。 对于 `SimpleTcpCluster` 类或者调用 `SimpleTcpCluster.send` 方法的对象，它们所发出的每一个消息上都附加着一个 `channelSendOptions` 标志。关于发送标志的描述可参见我们的 javadoc 文档。 `DeltaManager` 使用 `SimpleTcpCluster.send` 方法发送信息，而备份管理器则直接通过 channel 来发送自身。

更多详细信息请参见集群配置参考文档。

```
          <Manager className="org.apache.catalina.ha.session.BackupManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"
                   mapSendOptions="6"/>
          <!--
          <Manager className="org.apache.catalina.ha.session.DeltaManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"/>
          -->
```

如果在 `<Context>` 元素中没有定义 manager，则以上可当做 manager 的配置模板。在 Tomcat 5.x 时期，每个标识为可分发（distributable）的 Web 应用都必须使用同样的 manager，而如今不同了，我们可以为每个应用定义一个 manager 类，从而在集群中混合多个 manager。显然，A 节点上的某个应用的所有 manager 必须与 B 节点上的同样应用的 manager 相同。如果没有为应用指定 manager，而且该应用被标识为 `<distributable/>`，Tomcat 就会采取这种 manager 配置，创建一个克隆该配置的 manager 实例。

更多详细信息请参见集群管理器文档。

```
<Channel className="org.apache.catalina.tribes.group.GroupChannel">
```

Channel 元素是 Tribes 架构的一个重要组成部分，Tribes 是 Tomcat 内部所使用的分组通信架构。Channel 元素封装了所有通信相关事项以及成员逻辑。

详情参见集群 Channel 文档。

```
            <Membership
className="org.apache.catalina.tribes.membership.McastService"
                        address="228.0.0.4"
                        port="45564"
                        frequency="500"
                        dropTime="3000"/>
```

成员关系（Membership）是通过组播来实现的。注意，如果你想将成员扩展到组播范围之外的某个点时，Tribes 现在已经能够支持使用 `StaticMembershipInterceptor` 的静态成员。 `address` 属性是所用的组播地址， `port` 是所用的组播端口号。这两项组合起来将集群隔离开。如果你希望一个 QA 集群和一个生产集群，最简单的方法就是将 QA 集群的组播地址和端口号不同于生产集群的组播地址和端口号组合。

成员组件将其自身的 TCP 地址和端口广播到其他节点处，从而使节点间的通信都可以通过 TCP 协议来完成。请注意被广播的 TCP 地址正是 `Receiver.address` 属性值。

详情参见集群成员文档。

```
<Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
address="auto"
port="5000"
selectorTimeout="100"
maxThreads="6"/>
```

在 Tribes 架构中，数据的发送与接收以及被拆分为两种功能性组件了。正如其名所示，Receiver 负责接收信息。由于 Tribes 与线程无关（其他架构也开始采用这一种常见改进了），该组件内部包含一个线程池，设定有 `maxThreads` 和 `minThreads` 两种参数。

`address` 参数值是主机地址，由成员组件广播到其他节点中。

关于更多详情，可参看Receiver 文档。

```
            <Sender
className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
              <Transport
className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
            </Sender>
```

Sender 组件负责将消息发送给其他节点。Sender 含有一个 shell 组件 `ReplicationTransmitter`，但真正所要完成的任务则是通过子组件 `Transport` 来完成的。由于 Tribes 支持一个 Sender 池，所以消息可以做到同步；如果使用的是 NIO Sender，你也可以并发地发送消息。

并发（Concurrently）意味着将同时有多个发送者对应着一条消息，并行（Parallel）则意味着同时有多个消息对应着多个发送者。详情请参考这篇文档。

```
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Intercept
or"/>
            <Interceptor
className="org.apache.catalina.tribes.group.interceptors.ThroughputInterceptor"/>
          </Channel>
```

Tribes 利用了一个堆栈传送消息。每个堆栈内的元素都被称为拦截器，跟 Tomcat 容器中的 valve 的作用差不多。使用拦截器，逻辑可被分成更容易管理的代码段。上面配置中的拦截器：

- `TcpFailureDetector` 通过 TCP 核实崩溃的节点。如果组播包丢失，该拦截器就会防止误报的情况出现，比如，某个正在运行的节点虽然活跃，但也被标示为已崩溃。
- `MessageDispatch15Interceptor` 分派消息到线程（线程池），异步发送消息。
- `ThroughputInterceptor` 输出对信息流量的简单统计。

请注意，拦截器的顺序很重要。在 server.xml 中定义的顺序正是它们出现在 channel 堆栈中的顺序。这种机制就像是链表，最前面的是第一个拦截器，末尾的是最后一个拦截器。更多详细资料，可参看这篇文档。

```
 <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"

filter=".*\.gif|.*\.js|.*\.jpeg|.*\.jpg|.*\.png|.*\.htm|.*\.html|.*\.css|.*\.txt"/>
```

集群使用 valve 来跟踪针对 Web 应用的请求。我们之前已经提到过 ReplicationValve 和 JvmRouteBinderValve。`<Cluster>` 元素本身并不是 Tomcat 管道的一部分，集群将 valve 添加到了它的父容器上，比如说 `<Cluster>` 元素被配置到 `<Engine>` 元素中，那么 valve 就会被加到 `<Engine>` 元素中。更多详情，请参考集群 valve 配置文档。

```
          <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                    tempDir="/tmp/war-temp/"
                    deployDir="/tmp/war-deploy/"
                    watchDir="/tmp/war-listen/"
                    watchEnabled="false"/>
```

默认的 Tomcat 集群支持耕种部署（farmed deployment），比如说集群可以在其他的节点上部署和取消部署应用。该组件的状态目前还不稳定，但我们很快就会解决这个问题。Tomcat 5.0 和 5.5 版本相比，在部署算法上有一点变化。组件的逻辑改变到部署目录必须与应用目录相匹配。

更多详情，请参考集群部署器文档。

```
          <ClusterListener
className="org.apache.catalina.ha.session.ClusterSessionListener"/>
        </Cluster>
```

因为 `SimpleTcpCluster` 本身既是 Channel 对象的发送者，又是接受者，所以组件可以将它们自身注册成 SimpleTcpCluster 的侦听器。上面这个侦听器 `ClusterSessionListener` 将侦听 `DeltaManager` 复制的消

息，并将会话变更应用到 manager 上，反过来应用到会话上。

更多详情，参看 集群侦听器文档。

集群架构

组件层级：

```
Server
|
Service
|
Engine
|  \
|  --- Cluster --*
|
Host
## |/      \
Cluster    Context(1-N)
|             \
|             -- Manager
|                   \
|                   -- DeltaManager
|                   -- BackupManager
## ||                       \
Channel                     \
----------------------------- \
|                            \
Interceptor_1 ..                \
|                                \
Interceptor_N                     \
-----------------------------      \
|          |          |             \
Receiver    Sender   Membership       \
-- Valve
|      \
|       -- ReplicationValve
|       -- JvmRouteBinderValve
|
-- LifecycleListener
|
-- ClusterListener
|      \
|       -- ClusterSessionListener
|
-- Deployer
\
-- FarmWarDeployer
```

工作原理

为了便于理解集群的工作机制，下面将通过一些实际情境来加深一下你的理解，我们只打算采用 2 个 Tomcat 实例：`Tomcat A` 和 `Tomcat B`。具体发生的事件流程为：

1. `Tomcat A` 启动。
2. `Tomcat A` 启动完毕后，`Tomcat B` 才启动。
3. `Tomcat A` 接收一个请求，创建了一个会话 `S1`。
4. `Tomcat A` 崩溃。
5. `Tomcat B` 接收到对会话 `S1` 的请求。
6. `Tomcat A` 启动。
7. `Tomcat A` 接收到一个请求，调用会话 `S1` 上的 `invalidate` 方法。
8. `Tomcat B` 接收到一个对新会话 `S2` 的请求。
9. `Tomcat A` 会话 `S2` 由于不活跃而超时。

介绍完了事件序列，下面详细剖析一下在会话复制代码中到底发生了什么。

1.**`Tomcat A`** 启动

Tomcat 使用标准启动顺序来启动。Host 对象创建好之后，会关联一个 Cluster 对象。在解析上下文时，如果 web.xml 中包含 distributable 元素，Tomcat 就会让 Cluster 类（在该例中是 `SimpleTcpCluster`）创建复制的上下文的管理器。启用了集群并在 web.xml 中设置了 distributable 元素后，Tomcat 会为该上下文创建一个 `DeltaManager`（而不是 `StandardManager`）。Cluster 类会启动一个成员服务（组播）和一个复制服务（TCP 单播）。下文将会介绍更多的架构细节。

2.**`Tomcat B`** 启动

Tomcat B 启动时，采取的顺序与 Tomcat A 基本一样。集群启动，建立成员（Tomcat A 与 Tomcat B）。Tomcat B 会请求集群中已有服务器（本例中是 Tomcat A）的会话状态。如果 Tomcat A 响应该请求，那么在 Tomcat B 开始侦听 HTTP 请求之前，Tomcat A 会将会话状态传到 Tomcat B那里；如果 Tomcat A 没有响应该请求，Tomcat 会等待 60 秒，超过这个时间之后，发出一个日志项。该会话状态会发送到每一个在 web.xml 中设置了 distributable 元素的应用。注意：为了有效地使用会话复制，所有的 Tomcat 实例都必须拥有相同的配置。

3.**`Tomcat A`** 接收一个请求，创建了一个会话 **`S1`**

Tomcat A 对发送给它的请求的处理方式，与没有会话复制时的处理方式完全相同。请求完成时会触发相应行为，`ReplicationValve` 会在响应返回用户之前拦截请求。如发现会话已经更改，则使用 TCP 将会话复制到 Tomcat B 上。一旦序列化的数据被转交给操作系统的 TCP 逻辑，请求就会重新通过 valve 管道返回给用户。对于每一个请求，都将复制所有的会话，这样做就有利于复制那些在会话中修改属性的代码，使其即使不必调用 `setAttribute` 或 `removeAttribute`，也能被复制。另外，使用 `useDirtyFlag` 配置参数也可以优化会话的复制次数。

4.**`Tomcat A`** 崩溃

当 Tomcat A 崩溃时，Tomcat B 会接到通知，得知 Tomcat A 已被移出集群，随即 Tomcat B 就在其成员列表中也将 Tomcat A 移除，Tomcat B 从而不再收到关于 Tomcat A 的任何通知。负载均衡器会把从 Tomcat A 发送给 Tomcat B 的请求重新定向，所有的会话都将保持现有的状态。

5.**`Tomcat B`** 接收到对会话 **`S1`** 的请求

毫无悬念，Tomcat B 会照处理其他请求的方式那样来处理该请求。

6.**`Tomcat A`** 启动

在 Tomcat A 开始接收新的请求之前，将会根据上面（1）（2）两条所所说明的启动序列来启动。Tomcat A 会加入集群，联系 Tomcat B 并获取所有的会话状态。一旦接收到会话状态，就会完成加载，并打开 HTTP/mod_jk 端

口。所以，除非 Tomcat A 从 Tomcat B 那里接收到了会话变更，否则没有发给 Tomcat A 的请求。

7.**Tomcat A** 接收到一个请求，调用会话 **S1** 上的 **invalidate** 方法

会拦截对 `invalidate` 的调用, 并且 `session` 会被加入失效会话队列。 在请求完成时，不会发送会话改变消息，而是发送一个“到期”消息给 Tomcat B，Tomcat B 也会让此会话失效。

8.**Tomcat B** 接收到一个对新会话 **S2** 的请求

同步骤 3。

9.**Tomcat A** 会话 **S2** 由于不活跃而超时

invalidate 调用会被拦截，当一个会话被用户标记失效时，该会话就会加入到无效会话队列。此时，失效的会话不会被复制，直到另一个请求通过系统并检查无效会话队列。

Membership 集群成员是通过非常简单的组播 ping 命令来实现的。每个 Tomcat 实例都会定期发送一个组播 ping，ping 消息中包含 Tomcat 实例自身的 IP 和配置的 TCP 监听端口。如果实例在一个给定的时间内没有收到这样的 ping 信息,就会认为那个成员已经崩溃了。非常简洁高效！当然，您需要在系统上启用广播。

TCP 复制 一旦收到一个多播 ping 包，在下一个复制请求时成员被添加到集群，发送实例将使用的主机和端口信息，以及建立TCP套接字。使用该套接字发送序列化的数据。之选择TCP套接字，是因为它内建有流量控制和保证发送的功能。所以发送的数据肯定会到达那里。

分布式的锁定与使用架构的页面**s** Tomcat 在跨集群同步不保持会话实例。这种逻辑的实现将是多开销和导致各种各样的问题。如果你的客户用同一个会话同时发送多个请求，那么最后的请求将会覆盖集群中的其他会话。

利用 JMX 监控集群

使用集群时，如何监控是一个重要课题。有些集群对象是 JMX MBean。

添加下列属性到启动脚本上。

```
set CATALINA_OPTS=\
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=%my.jmx.port% \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false
```

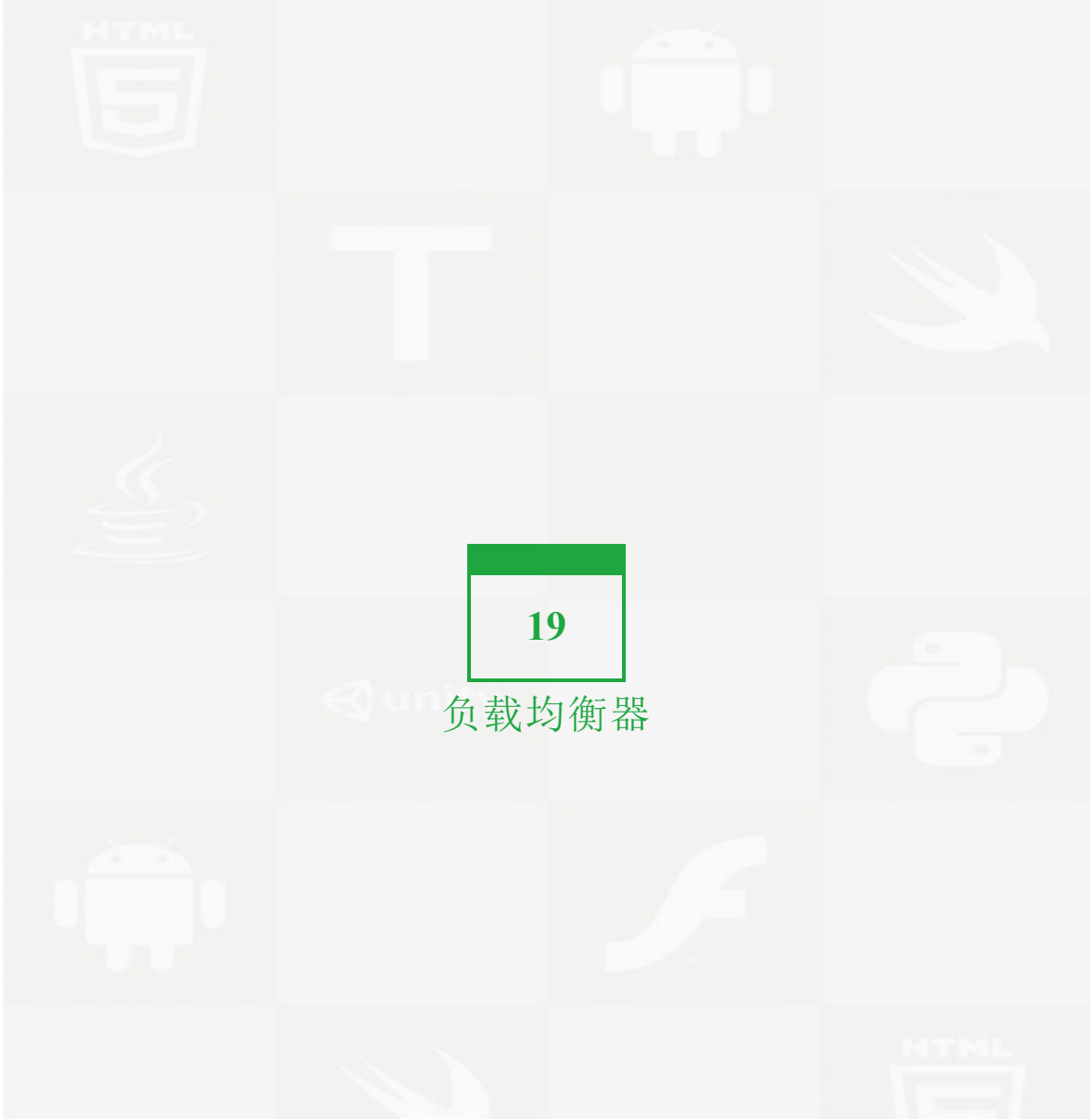
下面是 Cluster 的 MBean 列表：

名称	描述	MBean 对象名-引擎
`Cluster`	完整的 cluster 元素	`type=Cluster`
`DeltaManager`	该管理器控制会话，并处理会话复制	`type=Manager,context host=${HOST}`
`FarmWarDeployer`	将一个应用部署到该集群的所有节点上。	目前不支持
`Member`	代表集群中的一个节点	`type=Cluster, compon name=${NODE_NAME}`
`ReplicationValve`	该 valve 控制到备份节点的会	`type=Valve,name=Repl`

	话复制	
`JvmRouteBinderValve`	将 Session ID 变为 tomcat 当前的 jvmroute 的集群回滚值	`type=Valve,name=JvmR context=${APP.CONTEX`

常见问题解答

请参看 FAQ：集群文档。



19

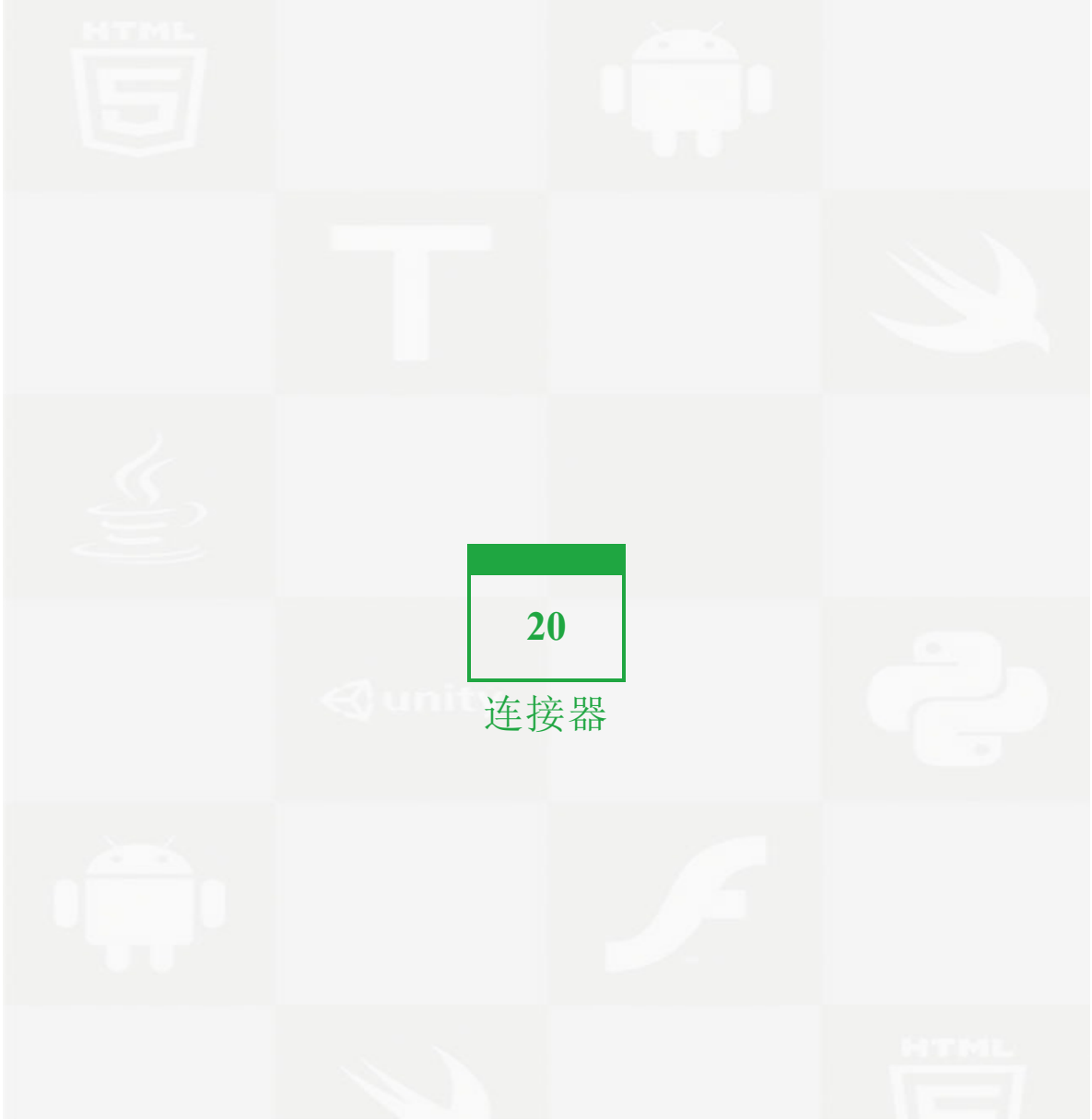
负载均衡器

使用 **JK 1.2.x** 原生连接器

请参考 JK 1.2.x 文档。

使用 **Apache HTTP Server 2.x**

请参阅 Apache HTTP Server 2.2 的 mod_proxy 文档。它能支持 HTTP 或 AJP 负载均衡。新版的 mod_proxy 也能适用于 Apache HTTP Server 2.0，但必须使用 Apache HTTP Server 2.2 的代码独立编译。



20

连接器

简介

选择适用于 Tomcat 的连接器是非常困难的。本文列出了目前版本的 Tomcat 所支持的连接器，可根据具体需要来选择使用。

HTTP

HTTP 连接器是 Tomcat 默认配置好的，可立即使用。该连接器能实现最低的延时以及最佳的整体性能。

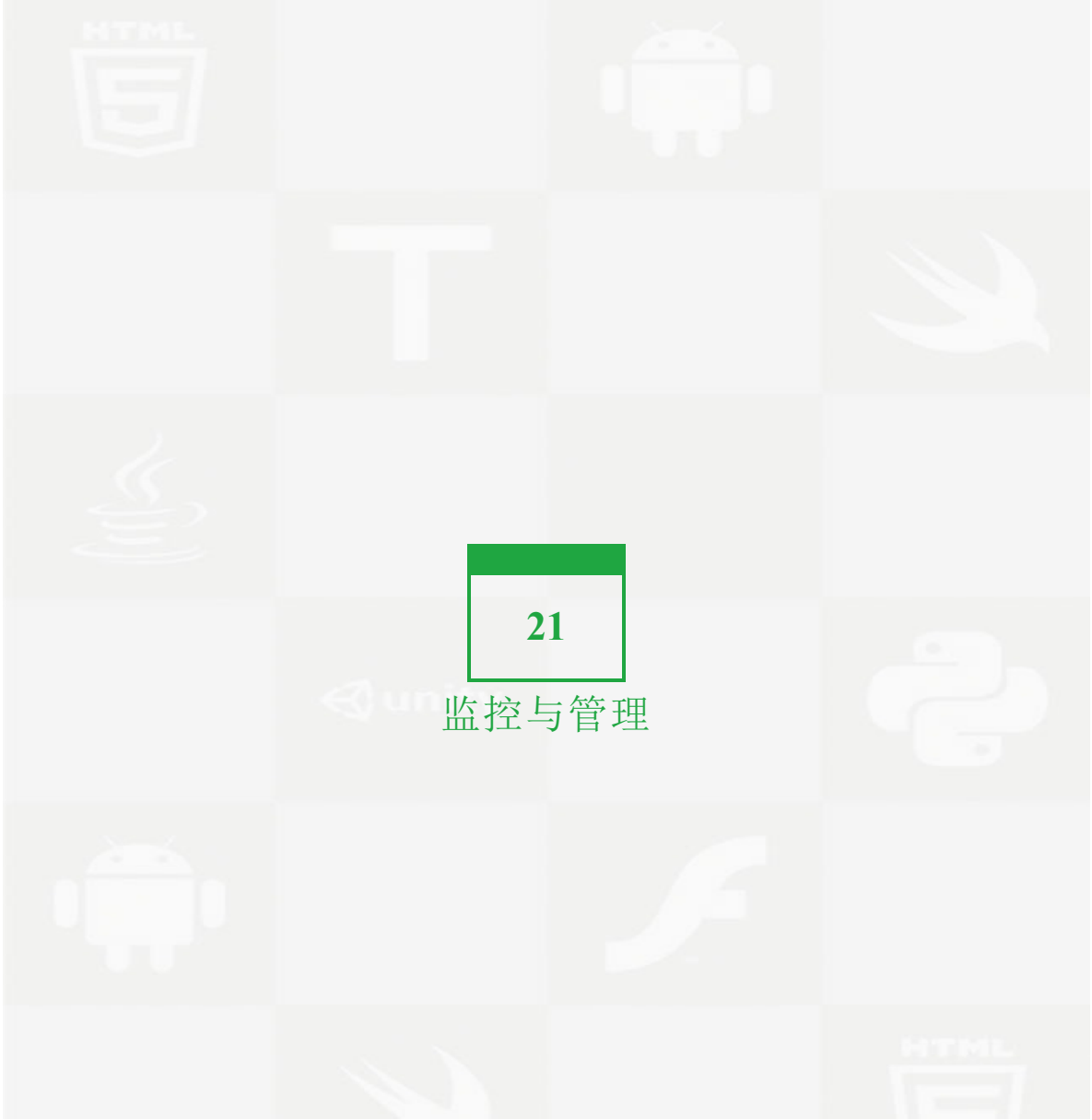
对于集群化来说，必须安装支持 **Web** 会话粘性的 HTTP 负载均衡器，以便将流量导引至多个 Tomcat 服务器上。Tomcat 支持将 mod_proxy 模块（可加载到 Apache HTTP server 2.0 中，到了 Apache HTTP server 2.2 时，成为默认包含的模块。）用作负载均衡器。不过要注意的是，HTTP 代理的性能往往要低于 AJP，所以 AJP 集群化才是首选方式。

AJP

在仅使用一个服务器的情况下，使用位于 Tomcat 实例之前的原生 Web 服务器，往往要比使用带有默认 HTTP 连接器的 Tomcat 要低效得多，即使当大部分 Web 应用都只是由静态文件构成时，情况依然是这样。但假如基于某种原因，必须要使用原生的 Web 服务器时，那么使用 AJP 连接器，就会比使用 HTTP 代理在性能上更加优越。从 Tomcat 的角度来看，AJP 集群无疑是最高效的。除了这一点之外，AJP 集群与 HTTP 集群在功能上是等同的。

这一版本的 Tomcat 所支持的原生连接器有：

- JK 1.2.x + 任何支持的服务器；
- Apache HTTP Server 2.x 上的 启用了 AJP 的 mod_proxy 模块（在 Apache HTTP Server 2.2 上已成为默认配置模块）。



21 监控与管理

简介

监控是系统管理中的重要环节。系统管理员的日常工作就包括：观察服务器的运行细节，获取统计数据，或者重新配置应用的某些内容。

启用 JMX 远程监控

注意：该配置只适用于需用远程监控 Tomcat 的情况，使用同样的用户在本地监控 Tomcat 则不需要这么配置。

Oracle 的网站上介绍了如何在 Java 6 上配置 JMX 远程：http://docs.oracle.com/javase/6/docs/technotes/guides/management/agent.html。

下面是在 Java 6 上的快速配置向导：

将下列参数添加到 Tomcat 的 `setenv.bat` 脚本（具体详细信息请查看 RUNNING.txt）。

注意：该语法格式适用于 Windows 系统。命令行只能写在同一行中，包装起来更具可读性。如果 Tomcat 以 Windows 服务的形式运行，使用它的系统配置对话设置该服务的 java 选项。对于 `UN\*X` 系统来说，要将命令行开头的 `"set "` 去掉。

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote
  -Dcom.sun.management.jmxremote.port=%my.jmx.port%
  -Dcom.sun.management.jmxremote.ssl=false
  -Dcom.sun.management.jmxremote.authenticate=false
```

1.如果需要授权，则添加并修改下列命令：

```
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.password.file=../conf/jmxremote.password
-Dcom.sun.management.jmxremote.access.file=../conf/jmxremote.access
```

2.编辑访问权限文件 `$CATALINA_BASE/conf/jmxremote.access`：

```
monitorRole readonly
controlRole readwrite
```

3.编辑密码文件 `$CATALINA_BASE/conf/jmxremote.password`：

```
monitorRole tomcat
controlRole tomcat
```

技巧：密码文件应该是只读的，并且只能被运行 Tomcat 的操作系统用户所访问。

注意：JSR 160 JMX 适配器在一个随机端口上打开了第二个数据通道。假如本地安装了防火墙，这就会出现问题。要想解决它，可以按照侦听器文档中介绍的方法，配置一个 `JmxRemoteLifecycleListener`。

利用 JMX 远程 Ant 任务来管理 Tomcat

为了简化 JMX 的用法，加入了一些可能会与 antlib 使用的一系列任务。

antlib: 将 catalina-ant.jar 从 `$CATALINA_HOME/lib` 复制到 `$ANT_HOME/lib`。

下面的例子展示了 JMX 存储器的用法。

注意：为了提高可读性，这里将 `name` 属性值予以包装。它必须写在同一行中，不允许带有空格。

```
<project name="Catalina Ant JMX"
      xmlns:jmx="antlib:org.apache.catalina.ant.jmx"
      default="state"
      basedir=".">
  <property name="jmx.server.name" value="localhost" />
  <property name="jmx.server.port" value="9012" />
  <property name="cluster.server.address" value="192.168.1.75" />
  <property name="cluster.server.port" value="9025" />

  <target name="state" description="Show JMX Cluster state">
    <jmx:open
      host="${jmx.server.name}"
      port="${jmx.server.port}"
      username="controlRole"
      password="tomcat"/>
    <jmx:get
      name=
"Catalina:type=IDataSender,host=localhost,
senderAddress=${cluster.server.address},senderPort=${cluster.server.port}"
      attribute="connected"
      resultproperty="IDataSender.backup.connected"
      echo="false"
    />
    <jmx:get
      name="Catalina:type=ClusterSender,host=localhost"
      attribute="senderObjectNames"
      resultproperty="senderObjectNames"
      echo="false"
    />
    <!-- get current maxActiveSession from ClusterTest application
       echo it to Ant output and store at
       property <em>clustertest.maxActiveSessions.orginal</em>
    -->
    <jmx:get
      name="Catalina:type=Manager,context=/ClusterTest,host=localhost"
      attribute="maxActiveSessions"
      resultproperty="clustertest.maxActiveSessions.orginal"
      echo="true"
    />
    <!-- set maxActiveSession to 100
    -->
    <jmx:set
      name="Catalina:type=Manager,context=/ClusterTest,host=localhost"
      attribute="maxActiveSessions"
      value="100"
      type="int"
    />
    <!-- get all sessions and split result as delimiter <em>SPACE</em> for easy
       access all session ids directly with Ant property sessions.[0..n].
```

```
    -->
    <jmx:invoke
      name="Catalina:type=Manager,context=/ClusterTest,host=localhost"
      operation="listSessionIds"
      resultproperty="sessions"
      echo="false"
      delimiter=" "
    />
    <!-- Access session attribute <em>Hello</em> from first session.
    -->
    <jmx:invoke
      name="Catalina:type=Manager,context=/ClusterTest,host=localhost"
      operation="getSessionAttribute"
      resultproperty="Hello"
      echo="false"
    >
      <arg value="${sessions.0}"/>
      <arg value="Hello"/>
    </jmx:invoke>
    <!-- Query for all application manager.of the server from all hosts
       and bind all attributes from all found manager MBeans.
    -->
    <jmx:query
      name="Catalina:type=Manager,*"
      resultproperty="manager"
      echo="true"
      attributebinding="true"
    />
    <!-- echo the create properties -->
<echo>
senderObjectNames: ${senderObjectNames.0}
IDataSender.backup.connected: ${IDataSender.backup.connected}
session: ${sessions.0}
manager.length: ${manager.length}
manager.0.name: ${manager.0.name}
manager.1.name: ${manager.1.name}
hello: ${Hello}
manager.ClusterTest.0.name: ${manager.ClusterTest.0.name}
manager.ClusterTest.0.activeSessions: ${manager.ClusterTest.0.activeSessions}
manager.ClusterTest.0.counterSend_EVT_SESSION_EXPIRED:
 ${manager.ClusterTest.0.counterSend_EVT_SESSION_EXPIRED}
manager.ClusterTest.0.counterSend_EVT_GET_ALL_SESSIONS:
 ${manager.ClusterTest.0.counterSend_EVT_GET_ALL_SESSIONS}
</echo>

  </target>

</project>
```

导入：利用 `<import file="${CATALINA.HOME}/bin/catalina-tasks.xml" />` 导入 JMX 存取器项目，利用 jmxOpen、jmxSet、jmxGet、jmxQuery、jmxInvoke、jmxEquals 和 jmxCondition 来引用任务。

JMXAccessorOpenTask - JMX 打开连接任务

属性列表

属性	描述
url	设定 JMX 连接 URL ——`service:jmx:rmi:///jndi/rmi://localho`
host	设定主机，缩短长的 URL 格式
port	设定远程连接端口
username	远程 JMX 连接用户名
password	远程 JMX 连接密码
ref	内部连接引用的名称。利用该属性，在同一个 Ant 项目中
echo	Echo 命令用途（用于访问分析或调试）
if	只有当指定名称的属性存在于当前项目时才执行
unless	只有当指定名称的属性不存在于当前项目时才执行

打开新的 JMX 连接的范例如下：

```
<jmx:open
  host="${jmx.server.name}"
  port="${jmx.server.port}"
/>
```

打开指定 URL 的 JMX 连接的范例，带有授权并存储在其他引用中：

```
<jmx:open
  url="service:jmx:rmi:///jndi/rmi://localhost:9024/jmxrmi"
  ref="jmx.server.9024"
  username="controlRole"
  password="tomcat"
/>
```

打开指定 URL 的 JMX 连接的范例，带有授权并存储在其他引用中，但是必须要求 *jmx.if* 属性存在，而 *jmx.unless* 属性不存在。

```
<jmx:open
  url="service:jmx:rmi:///jndi/rmi://localhost:9024/jmxrmi"
  ref="jmx.server.9024"
  username="controlRole"
```

```
    password="tomcat"
    if="jmx.if"
    unless="jmx.unless"
  />
```

注意：*jmxOpen* 任务中所有属性也存在于其他所有任务和条件中。

JMXAccessorGetTask: 获取属性值的 Ant 任务

属性列表

属性	描述
`name`	完全限定的 JMX ObjectName ——*Catalina:type=Server*
`attribute`	已有的 MBean 属性（参看上文介绍的 Tomcat MBean 描述）
`ref`	JMX 连接引用
`echo`	Echo 命令用途（访问与结果）
`resultproperty`	在该项目属性中保存结果
`delimiter`	用分隔符（`java.util.StringTokenizie` 分隔结果，使用 resultproperty 作为前来保存令牌
`separatearrayresults`	返回值为数组时，将结果保存为属性（*$resultproperty.[0..N]* 和 *$resultproperty.length*）

从默认的 JMX 连接中获取远程 MBean 属性：

```
<jmx:get
  name="Catalina:type=Manager,context=/servlets-examples,host=localhost"
  attribute="maxActiveSessions"
  resultproperty="servlets-examples.maxActiveSessions"
/>
```

获取结果数组，并将其分隔成独立的一些属性：

```
<jmx:get
    name="Catalina:type=ClusterSender,host=localhost"
    attribute="senderObjectNames"
    resultproperty="senderObjectNames"
/>
```

访问 senderObjectNames 属性：

```
${senderObjectNames.length} give the number of returned sender list.
${senderObjectNames.[0..N]} found all sender object names
```

获取连接的 IDataSender 属性（只有在配置了集群时）。

注意：这里为了可读性，将 `name` 属性加以包装。代码应该位于同一行中，并且不含有空格。

```
  <jmx:query
    failonerror="false"
    name="Catalina:type=Cluster,host=${tomcat.application.host}"
    resultproperty="cluster"
  />
  <jmx:get
    name=
"Catalina:type=IDataSender,host=${tomcat.application.host},
senderAddress=${cluster.backup.address},senderPort=${cluster.backup.port}"
    attribute="connected"
    resultproperty="datasender.connected"
    if="cluster.0.name" />
```

JMXAccessorSetTask:设定属性值的 Ant 任务

属性列表

属性	说明	默认值
`name`	完全限定的 JMX ObjectName ——`Catalina:type=Server`	-
`attribute`	已有的 MBean 属性（详情参见上文介绍的 Tomcat MBean 说明）	-
`value`	设定为属性的值	-
`type`	属性类型	`java.lang.St`
`ref`	JMX 连接引用	`jmx.server`
`echo`	Echo 命令用途（访问与结果）	`false`

设定远程 MBean 属性值的范例如下：

```
<jmx:set
  name="Catalina:type=Manager,context=/servlets-examples,host=localhost"
  attribute="maxActiveSessions"
  value="500"
  type="int"
/>
```

JMXAccessorInvokeTask: 调用 MBean 操作的 Ant 任务

属性列表

属性	说明
`name`	完全限定的 JMX ObjectName ——`Catalina:type=Server`
`operation`	已有的 MBean 操作（funcspecs/fs-adn opers.html）
`ref`	JMX 连接引用
`echo`	Echo 命令用途（访问与结果）
`resultproperty`	在这一项目属性中保存结果
`delimiter`	用分隔符（`java.util.StringTokenizie` 分隔结果，使用 resultproperty 作为前来保存令牌
`separatearrayresults`	返回值为数组时，将结果保存为属性（*$resultproperty.[0..N]* 和 *$resultproperty.length*）

停止应用：

```
<jmx:invoke
  name="Catalina:type=Manager,context=/servlets-examples,host=localhost"
  operation="stop"/>
```

可以在 *${sessions.[0..N}* 属性中找到 sessionid，然后利用 `${sessions.length}` 属性来访问计数。

获取所有 sessionid 的范例如下：

```
<jmx:invoke
  name="Catalina:type=Manager,context=/servlets-examples,host=localhost"
  operation="listSessionIds"
  resultproperty="sessions"
  delimiter=" "
/>
```

现在你可以在 *${sessions.[0..N}* 属性中找到 sessionid，然后利用 `${sessions.length}` 属性来访问计数。

从 ${sessionid.0} 会话中获取远程 MBean 会话属性：

```
 <jmx:invoke
   name="Catalina:type=Manager,context=/ClusterTest,host=localhost"
   operation="getSessionAttribute"
   resultproperty="hello">
    <arg value="${sessionid.0}"/>
    <arg value="Hello" />
 </jmx:invoke>
```

在虚拟主机 *localhost* 上创建新的访问日志记录器值：

```
<jmx:invoke
        name="Catalina:type=MBeanFactory"
        operation="createAccessLoggerValve"
        resultproperty="accessLoggerObjectName"
>
    <arg value="Catalina:type=Host,host=localhost"/>
</jmx:invoke>
```

现在可以利用 *${accessLoggerObjectName}* 属性上存储的名称找到新的 MBean 了。

JMXAccessorQueryTask: 查询 MBean 的 Ant 任务

属性列表：

属性	描述
`name`	JMX ObjectName 查询字符串——*Catalina:type=Manager,**
`ref`	JMX 连接引用
`echo`	Echo 命令用途（访问及结果）
`resultproperty`	将项目属性名做为前缀加到所有已建 MBean 上（`mbeans.[0..N].objectname`）
`attributebinduing`	除了 `name` 之外，绑定所有的 MBea 性
`delimiter`	用分隔符（`java.util.StringTokenizie` 分隔结果，使用 resultproperty 作为前来保存令牌
`separatearrayresults`	返回值为数组时，将结果保存为属性（*$resultproperty.[0..N]* 和 *$resultproperty.length*）

从所有的服务及主机中获取所有的 Manager ObjectName：

```
<jmx:query
  name="Catalina:type=Manager,*
  resultproperty="manager" />
```

现在，在 *${manager.[0..N].name}* 属性上可以找到 Session Manager，利用 ${manager.length} 属性来访问结果对象计数器。

从 *servlet-examples* 程序中获取 Manager，并绑定所有的 MBean 属性：

```
<jmx:query
  name="Catalina:type=Manager,context=/servlet-examples,host=localhost*"
  attributebinding="true"
  resultproperty="manager.servletExamples" />
```

现在我们可以在 ${manager.servletExamples.0.name} 属性中找到 manager，并利用 ${manager.servletExamples.0.

[manager attribute names]} 访问该 manager 的所有属性。MBean 的结果对象计数器被保存在 ${manager.length} 属性中。

在下面范例中，从服务器中获取所有的 MBean，并保存在外部的 XML 属性文件中。

```
<project name="jmx.query"
            xmlns:jmx="antlib:org.apache.catalina.ant.jmx"
            default="query-all" basedir=".">
<property name="jmx.host" value="localhost"/>
<property name="jmx.port" value="8050"/>
<property name="jmx.username" value="controlRole"/>
<property name="jmx.password" value="tomcat"/>

<target name="query-all" description="Query all MBeans of a server">
  <!-- Configure connection -->
  <jmx:open
    host="${jmx.host}"
    port="${jmx.port}"
    ref="jmx.server"
    username="${jmx.username}"
    password="${jmx.password}"/>

  <!-- Query MBean list -->
  <jmx:query
    name="*:*"
    resultproperty="mbeans"
    attributebinding="false"/>

  <echoproperties
    destfile="mbeans.properties"
    prefix="mbeans."
    format="xml"/>

  <!-- Print results -->
  <echo message=
    "Number of MBeans in server ${jmx.host}:${jmx.port} is ${mbeans.length}"/>
</target>
</project>
```

现在就可以在 mbeans.properties 文件中找到所有的 MBean 了。

JMXAccessorCreateTask: 远程创建 MBean 的 Ant 任务

属性列表

属性	描述
`name`	完全限定的 JMX ObjectName——*Catalina:type=M*
`className`	已有的 MBean 完全限定的类名（参见上文的 Tom
`classLoader`	服务器或 Web 应用类加载器的 ObjectName（*Catalina:type=ServerClassLoader,name=[server,c* 或 *Catalina:type=WebappClassLoader,context=/myapp*
`ref`	JMX 连接引用
`echo`	Echo 命令用途（访问及结果）

创建远程 MBean 的范例如下：

```
<jmx:create
  ref="${jmx.reference}"
  name="Catalina:type=MBeanFactory"
  className="org.apache.commons.modeler.BaseModelMBean"
  classLoader="Catalina:type=ServerClassLoader,name=server">
  <arg value="org.apache.catalina.mbeans.MBeanFactory" />
</jmx:create>
```

警告：许多 Tomcat MBean 一经创建就没有与父级连接。Valve、集群以及Realm 的 MBean 都不会自动与父级相连。作为替代，可以使用 MBeanFactory 来创建操作。

JMXAccessorUnregisterTask: 远程注销 MBean Ant 任务

属性列表

属性	描述	默认值
`name`	完全限定的 JMX ObjectName ——*Catalina:type=MBeanFactory*	=
`ref`	JMX 连接引用	`jmx.server`
`echo`	Echo 命令使用（访问及结果）	`false`

注销远程 MBean 范例如下：

```
<jmx:unregister
  name="Catalina:type=MBeanFactory"
/>
```

警告：许多 Tomcat MBean 都无法注销。MBean 无法从其父级脱离。可以使用 MBeanFactory 来移除操作。

JMXAccessorCondition: 表达条件

属性列表

属性	描述
`url`	设定 JMX 连接 URL ——*service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrr*
`host`	设定主机，将非常长的 URL 格式予以缩短
`port`	设定远程连接端口
`username`	远程 JMX 连接用户名
`password`	远程 JMX 连接密码
`ref`	内部连接引用名称。利用这一属性，可以在同一个 Ant 项目中配置多个连接。
`name`	完全限定的 JMX ObjectName ——*Catalina:type=Server*
`echo`	Echo 命令使用（访问及结果）
`if`	只有当给定名称的属性存在于当前项目中才执行
`unless`	只有当给定名称的属性不存在于当前项目中才执行
`value` （必须）	操作的第二个参数
`type`	表达操作的值类型（支持 *long* 和 *double*）
`operation`	提供以下操作 • `==` 等于 • `!=` 不等于 • `>` 大于（ `&gt` ） • `>=` 大于或等于（ `>=` ） • `<` 小于（ `&lt` ）

- `<=` 小于或等于（ `<=` ）

等待服务器连接，集群备份节点可访问。

```
<target name="wait">
  <waitfor maxwait="${maxwait}" maxwaitunit="second"
timeoutproperty="server.timeout" >
    <and>
      <socket server="${server.name}" port="${server.port}"/>
      <http url="${url}"/>
      <jmx:condition
        operation="=="
        host="localhost"
        port="9014"
        username="controlRole"
        password="tomcat"
        name=
"Catalina:type=IDataSender,host=localhost,senderAddress=192.168.111.1,senderPort=90
25"
        attribute="connected"
        value="true"
      />
    </and>
  </waitfor>
  <fail if="server.timeout" message="Server ${url} don't answer inside ${maxwait}
sec" />
  <echo message="Server ${url} alive" />
</target>
```

JMXAccessorEqualsCondition: MBean Ant 条件对等

属性列表

属性	描述
`url`	设定 JMX 连接 URL ——*service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi*
`host`	设定主机，将非常长的 URL 格式予以缩短
`port`	设定远程连接端口
`username`	远程 JMX 连接用户名
`password`	远程 JMX 连接密码
`ref`	内部连接引用名称。利用这一属性，可以在同一个 Ant 项目中配置多个连接。
`name`	完全限定的 JMX ObjectName ——*Catalina:type=Server*
`echo`	Echo 命令使用（访问及结果）

等待服务器连接，集群备份节点可访问。

```
<target name="wait">
  <waitfor maxwait="${maxwait}" maxwaitunit="second"
timeoutproperty="server.timeout" >
    <and>
      <socket server="${server.name}" port="${server.port}"/>
      <http url="${url}"/>
      <jmx:equals
        host="localhost"
        port="9014"
        username="controlRole"
        password="tomcat"
        name=
"Catalina:type=IDataSender,host=localhost,senderAddress=192.168.111.1,senderPort=90
25"
        attribute="connected"
        value="true"
      />
    </and>
  </waitfor>
  <fail if="server.timeout" message="Server ${url} don't answer inside ${maxwait}
sec" />
```

```
  <echo message="Server ${url} alive" />
</target>
```

使用 **JMXProxyServlet**

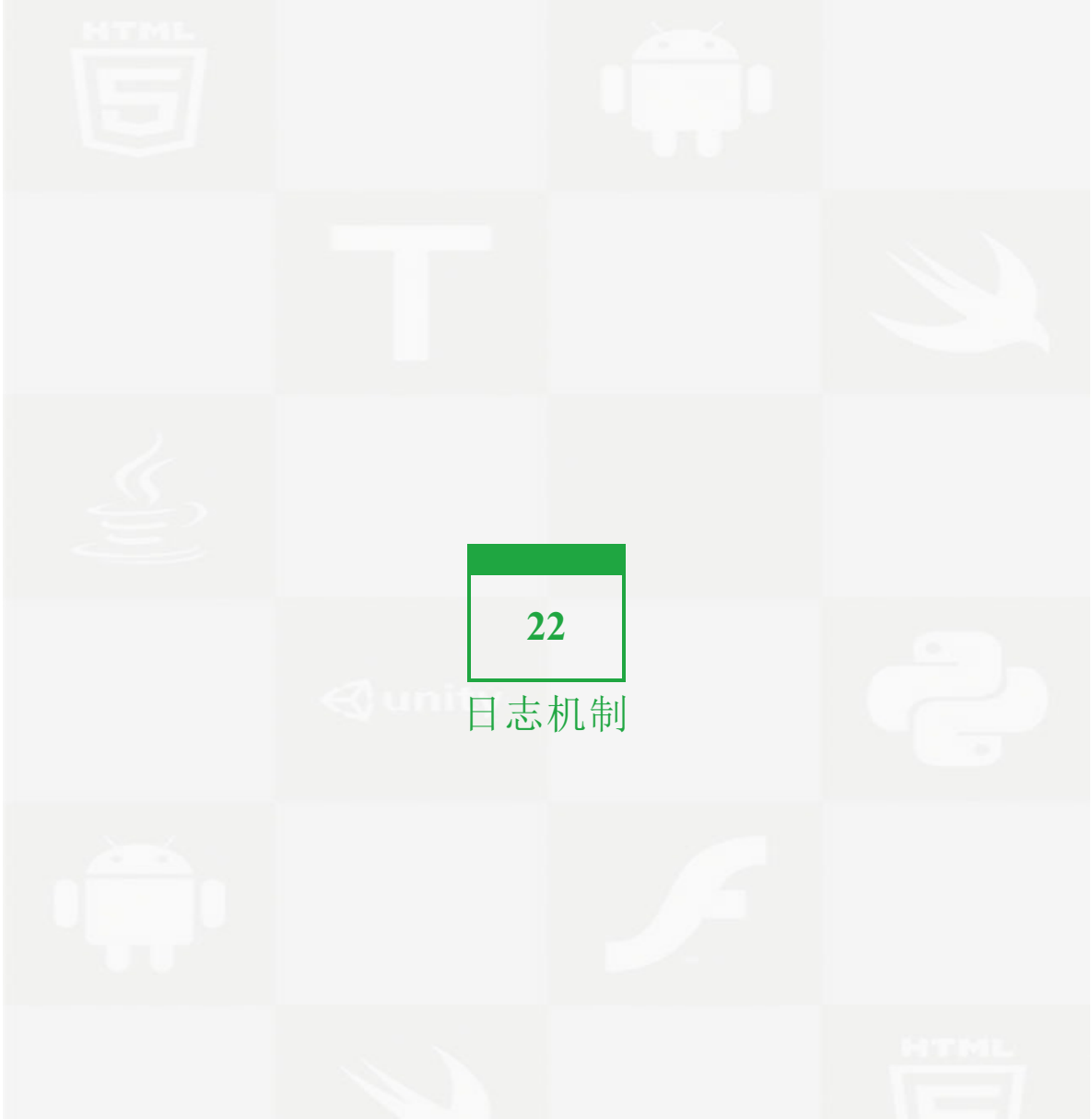
Tomcat 为使用远程（或者甚至本地的）JMX 连接提供了一个替代方案：Tomcat 的 JMXProxyServlet，但它仍能让你访问 JMX 所提供的任何内容。

JMXProxyServlet 允许客户端通过 HTTP 接口来发送 JMX 查询。相比直接从客户端程序使用 JMX 来说，该技术具有以下优势：

- 无需加载完整的 JVM 并执行远程 JMX 连接，只需从服务器上请求一小块数据即可。
- 无需了解处理 JMX 连接的方式。
- 无需任何复杂的配置。
- 无需用 Java 来编写客户端程序。

常见的服务器监控软件（比如 Nagios 或 Ichinga）中都存在过度使用 JMX 的问题：如果想通过 JMX 监控 10 项，就必须启动 10 个 JVM，保持 10 个 JMX 连接，每过几分钟就要将它们全部关闭。有了 JMXProxyServlet，利用 10 个 HTTP 连接就能搞定了。

关于 JMXProxyServlet 的详细说明，可查阅 Tomcat Manager。



22 日志机制

简介

Tomcat 的内部日志使用 JULI 组件，这是一个 Apache Commons 日志的重命名的打包分支，默认被硬编码，使用 `java.util.logging` 架构。这能保证 Tomcat 内部日志与 Web 应用的日志保持独立，即使 Web 应用使用的是 Apache Commons Logging。

假如想用另外的日志框架来替换 Tomcat 的内部日志系统，那么就必须采用一种能够保持完整的 Commons 日志机制的 JULI 实现，用它来替换通过硬编码使用 `java.util.logging` 的 JULI 实现。通常这种替代实现都是以额外组件的形式出现的。利用 Log4j 框架用于 Tomcat 内部日志的配置如下文所示。

在 Apache Tomcat 上运行的 Web 应用可以使用：

- 任何自选的日志框架。
- 系统日志 API，`java.util.logging`。
- Java Servlets 规范所提供的日志 API，`javax.servlet.ServletContext.log(...)`。

各个应用可以使用不同的日志框架，详情参见类加载器。`java.util.logging` 则是例外。如果日志库直接或间接地用到了这一 API，那么 Web 应用就能共享使用它的元素，因为该 API 是由系统类加载器所加载的。

Java 日志 API——java.util.logging

Apache Tomcat 本身已经实现了 `java.util.logging` API 的几个关键元素。这种实现就是 JULI。其中的关键组件是一个自定义的 LogManager 实现，它能分辨运行在 Tomcat 上的不同 Web 应用（以及它们所用的不同的类加载器），还能针对每一应用进行私有的日志配置。另外，当 Web 应用没能从内存中加载时，Tomcat 会给予它相应通知，从而清除相应的引用类，防止内存泄露。

在启动 Java 时，通过提供特定的系统属性，可以启用 `java.util.logging` 实现。Apache Tomcat 启动脚本可以实现这个操作，但如果使用不同工具来运行 Tomcat（比如 jsvc，或者从某个 IDE 中运行 Tomcat），就必须自己来启用实现。

关于 `java.util.logging` 实现的详细情况可以查阅 JDK 文档，具体位于 `java.util.logging` 包的相关 javadoc 页面中。

关于 Tomcat JULI 的详细介绍见下文。

Servlets logging API

Tomcat 内部日志能够处理对 `javax.servlet.ServletContext.log(...)` 的调用，从而写入日志消息。这种消息都被记录到一种特定类别中，命名方式如下：

```
org.apache.catalina.core.ContainerBase.[${engine}].[${host}].[${context}]
```

这种日志是依照 Tomcat 日志配置而执行的，无法在 Web 应用中重写。

Servlets logging API 的问世要先于 Java 所提供的 `java.util.logging` API，所以，它无法提供太多的选项，比如无法用它来控制日志级别。然而需要注意的是，在 Tomcat 实现中， 对 `ServletContext.log(String)` 和 `GenericServlet.log(String)` 的调用都被记录在 INFO 级别。对 `ServletContext.log(String, Throwable)` 或 `GenericServlet.log(String, Throwable)` 的调用都被记录在 ERROR 级别。

Console

在 UNIX 系统下运行 Tomcat 时，控制台输出经常会重定向到 `catalina.out` 的文件中。通过一个环境变量，

可以配置该文件（参见启动脚本）。

写入 `System.err/out` 的任何内容都会被 `catalina.out` 文件所捕获。这些内容可能包括：

- 由 `java.lang.ThreadGroup.uncaughtException(..)` 所输出的未捕获异常。
- 线程转储，如果通过系统信号来请求它们。

在 Windows 上以服务形式运行时，控制台输出也会被捕获及重定向，但文件名有所不同。

Tomcat 默认的日志配置会将同样的消息写入控制台和一个日志文件中。这一特点非常有利于使用 Tomcat 进行开发，但往往并不适用于生产环境。

老的应用可能还在使用 `System.out` 或 `System.err`，可以通过在 Context 元素上设置 `swallowOutput` 属性来调整。如该属性设为 `true`，那么在请求阶段对 `System.out/err` 的调用就会被拦截，它们的输出也会通过 `javax.servlet.ServletContext.log(...)` 调用反馈给日志系统。

注意：`swallowOutput` 虽然是一个小技巧，但还是有局限性的：它需要直接调用 `System.out/err`，并且要在请求处理周期内完成。而且，它可能还并不适用于应用所创建的其他线程。不能将其用于拦截本身写入系统流的日志框架（它们可能早先已经启动，并且在重定向发生前就已经获取了对流的直接引用）。

Acces 日志

Access 日志功能相近，但还是有所不同。它是一个 `Valve`，使用自包含的逻辑来编写日志文件。访问日志的基本需求是以较低开销处理大型连续数据流，所以只能使用 Commomns Logging 来处理自身的调试消息。这种实现方法避免了额外的开销，并且可能具有较复杂的配置。请参考 Valves 文档了解更多配置详情，其中包含了各种报告格式。

使用 java.util.logging（默认）

JDK 所提供的默认 java.util.logging 实现功能太过局限，所以根本没有什么使用价值。其关键局限在于不能实现针对每一应用进行日志记录，因为配置是针对每一 VM 的。所以按照默认配置，Tomcat 会用 JULI 这种非常适用于容器的实现来代替默认的 LogManager 实现，从而避免了 LogManager 的缺点。

跟标准 JDK 的 `java.util.logging` 一样，JULI 也支持同样的配置机制，或者使用编程方式，或者指定属性值。它与 `java.util.logging` 的不同在于，它可以分别设置每一个类加载器属性文件（能够启用简单的、便于重新部署的应用配置），属性文件还支持扩展构造，能够更加自由地定义 handle 并将其指定给 logger。

JULI 是默认启用的，除了普通的全局 java.util.logging 配置之外，它支持每个类加载器配置。这意味着可以在下列层级来配置日志：

- 全局范围。`${catalina.base}/conf/logging.properties` 文件。该文件通过由启动脚本设置的系统属性 `java.util.logging.config.file` 来指定。如果它不可读或没有配置，默认采用 JRE 中的 `${java.home}/lib/logging.properties` 文件。
- 在 Web 应用范围内。该文件为 `WEB-INF/classes/logging.properties`。

JRE 中默认的 `logging.properties` 指定了 `ConsoleHandler`，用于将日志输出至 `System.err`。Tomcat 中默认的 `conf/logging.properties` 也添加了几个能够写入文件的 `FileHandlers`。

handler 的日志级别容差值默认为 INFO，取值范围为：SEVERE、WARNING、INFO、CONFIG、FINE、FINER、FINEST 或 ALL。你也可以从特殊的包中收集日志，然后为这种日志指定相应的级别。

为了启用 部分 Tomcat 内部的调试日志功能，应该配置适合的 logger 和 handle 来使用 `FINEST` 或 `ALL` 级别。比如：

```
org.apache.catalina.session.level=ALL
java.util.logging.ConsoleHandler.level=ALL
```

当启用调试日志功能时，建议将范围尽量缩小，因为该功能会产生大量信息。

JULI 所使用的配置与纯 `java.util.logging` 所支持的配置基本相同，只不过使用了一些扩展，以便更灵活地配置 logger 和 handler。主要的差别在于：

- handler 名称前可以加上前缀，所以同一类可以实例化出多个 handler。前缀是一个以数字开头的字符串，并以 `.` 结尾。比如 `22foobar.` 就是个有效的前缀。
- 系统属性

还有一些额外的实现类，它们可以与 Java 所提供的类一起使用。在这些类中，最著名的就是 `org.apache.juli.FileHandler`。

`org.apache.juli.FileHandler` 支持日志缓存。日志缓存默认是没有启用的。使用 handler 的 `bufferSize` 属性可以配置它：属性值为 `0` 时，代表使用系统默认的缓存（通常使用 8k 缓存）；属性值小于 0 时，将在每个日志写入上强制使用 writer flush（将缓存区中的数据强制写出到系统输出）功能；属性值大于 0 时，则使用带有定义值的 BufferedOutputStream 类——但要注意的是，这也将应用于系统默认的缓存。

以下是一个 `$CATALINA_BASE/conf` 中的 `logging.properties` 文件：

```
handlers = 1catalina.org.apache.juli.FileHandler, \
           2localhost.org.apache.juli.FileHandler, \
           3manager.org.apache.juli.FileHandler, \
           java.util.logging.ConsoleHandler

.handlers = 1catalina.org.apache.juli.FileHandler, java.util.logging.ConsoleHandler

 ############################################################

 # Handler specific properties.
```

```
 # Describes specific configuration info for Handlers.

 ############################################################


1catalina.org.apache.juli.FileHandler.level = FINE
1catalina.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
1catalina.org.apache.juli.FileHandler.prefix = catalina.

2localhost.org.apache.juli.FileHandler.level = FINE
2localhost.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
2localhost.org.apache.juli.FileHandler.prefix = localhost.

3manager.org.apache.juli.FileHandler.level = FINE
3manager.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
3manager.org.apache.juli.FileHandler.prefix = manager.
3manager.org.apache.juli.FileHandler.bufferSize = 16384

java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter


 ############################################################

 # Facility specific properties.

 # Provides extra control for each logger.

 ############################################################


org.apache.catalina.core.ContainerBase.[Catalina].[localhost].level = INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].handlers = \
   2localhost.org.apache.juli.FileHandler

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].level =
INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].handlers =
\
   3manager.org.apache.juli.FileHandler

 # For example, set the org.apache.catalina.util.LifecycleBase logger to log

 # each component that extends LifecycleBase changing state:

 #org.apache.catalina.util.LifecycleBase.level = FINE
```

下例是一个用于 servlet-examples 应用的 `WEB-INF/classes` 中的 `logging.properties` 文件：

```
handlers = org.apache.juli.FileHandler, java.util.logging.ConsoleHandler

 ############################################################

 # Handler specific properties.

 # Describes specific configuration info for Handlers.

 ############################################################


org.apache.juli.FileHandler.level = FINE
```

```
org.apache.juli.FileHandler.directory = ${catalina.base}/logs
org.apache.juli.FileHandler.prefix = ${classloader.webappName}.

java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

1. 文档引用

查看下列资源获取额外的详细信息：

- org.apache.juli 包的相关 Tomcat 文档。
- java.util.logging 包的 Oracle Java 6 文档。

2. 生产环境使用中的注意事项

可能需要注意以下方面：

- 将 `ConsoleHandler` 从配置中移除。默认（多谢 `.handlers` 设置）日志会使用 `FileHandler` 和 `ConsoleHandler`。后者的输出经常会被捕获到一个文件中，比如 `catalina.out`。从而导致同一消息可能生成了两个副本。
- 对于不使用的应用(比如 `host-manager`)，可以考虑将 `FileHandlers` 移除。
- handler 默认使用系统缺省编码来写入日志文件，通过 `encoding` 属性可以修改设置，详情查看相关的 javadoc 文档。
- 配置 Access log 。

使用 Log4j

前面介绍了用于 Tomcat 内部日志的 java.util.logging，接下来本部分内容介绍如何通过配置 Tomcat 使用 log4j。

注意：当你想重新配置 Tomcat 以便利用 log4j 来进行自身日志记录时，下面的步骤都是必需的；而当你只是想在自己的 Web 应用上使用 log4j 时，这些步骤则不是必需的。在后一种情况下，只需将 `log4j.jar` 和 `log4j.properties` 放到 Web 应用的 `WEB-INF/lib` 和 `WEB-INF/classes` 中即可。

通过下列步骤可配置 log4j 输出 Tomcat 的内部日志：

1.创建一个包含下列配置的 `log4j.properties` 文件，将其保存到 `$CATALINA_BASE/lib`。

```
    log4j.rootLogger = INFO, CATALINA

    # Define all the appenders
    log4j.appender.CATALINA = org.apache.log4j.DailyRollingFileAppender
    log4j.appender.CATALINA.File = ${catalina.base}/logs/catalina
    log4j.appender.CATALINA.Append = true
    log4j.appender.CATALINA.Encoding = UTF-8
    # Roll-over the log once per day
    log4j.appender.CATALINA.DatePattern = '.'yyyy-MM-dd'.log'
    log4j.appender.CATALINA.layout = org.apache.log4j.PatternLayout
    log4j.appender.CATALINA.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

    log4j.appender.LOCALHOST = org.apache.log4j.DailyRollingFileAppender
    log4j.appender.LOCALHOST.File = ${catalina.base}/logs/localhost
    log4j.appender.LOCALHOST.Append = true
    log4j.appender.LOCALHOST.Encoding = UTF-8
    log4j.appender.LOCALHOST.DatePattern = '.'yyyy-MM-dd'.log'
    log4j.appender.LOCALHOST.layout = org.apache.log4j.PatternLayout
    log4j.appender.LOCALHOST.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

    log4j.appender.MANAGER = org.apache.log4j.DailyRollingFileAppender
    log4j.appender.MANAGER.File = ${catalina.base}/logs/manager
    log4j.appender.MANAGER.Append = true
    log4j.appender.MANAGER.Encoding = UTF-8
    log4j.appender.MANAGER.DatePattern = '.'yyyy-MM-dd'.log'
    log4j.appender.MANAGER.layout = org.apache.log4j.PatternLayout
    log4j.appender.MANAGER.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

    log4j.appender.HOST-MANAGER = org.apache.log4j.DailyRollingFileAppender
    log4j.appender.HOST-MANAGER.File = ${catalina.base}/logs/host-manager
    log4j.appender.HOST-MANAGER.Append = true
    log4j.appender.HOST-MANAGER.Encoding = UTF-8
    log4j.appender.HOST-MANAGER.DatePattern = '.'yyyy-MM-dd'.log'
    log4j.appender.HOST-MANAGER.layout = org.apache.log4j.PatternLayout
    log4j.appender.HOST-MANAGER.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

    log4j.appender.CONSOLE = org.apache.log4j.ConsoleAppender
    log4j.appender.CONSOLE.Encoding = UTF-8
    log4j.appender.CONSOLE.layout = org.apache.log4j.PatternLayout
    log4j.appender.CONSOLE.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

    # Configure which loggers log to which appenders
    log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost] =
INFO, LOCALHOST
    log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager] =\
  INFO, MANAGER
```

```
    log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager] =\
  INFO, HOST-MANAGER
```

2.下载 log4j(Tomcat 需要 1.2.x 版本)。

3.下载或构建 `tomcat-juli.jar` 和 `tomcat-juli-adapters.jar`，以便作为 Tomcat 的额外组件使用。详情参考 Additional Components documentation。

```
`tomcat-juli.jar` 跟默认的版本不同。它包含所有的 Commons Logging 实现，从而能够发现 log4j 并配置自身。
```

4.如果希望全局性地使用 log4j，则如下配置 Tomcat：
- 将 `log4j.jar` 和 `tomcat-juli-adapters.jar` 从 extras 中放入 `$CATALINA HOME/lib` 中。
- 用 extras 中的 `tomcat-juli.jar` 替换 `$CATALINA_HOME/bin/tomcat-juli.jar`。

5.如果是利用独立的 `$CATALINA_HOME` 和 `$CATALINA_BASE` 来运行 Tomcat，并想在一个 `$CATALINA_BASE` 中配置使用 log4j，则需要：
- 创建 `$CATALINA BASE/bin` 和 `$CATALINA BASE/lib` 目录——如果它们不存在的话。
- 将 extras 中的 `log4j.jar` 与 `tomcat-juli-adapters.jar` 从 extras 放入 `$CATALINA_BASE/lib` 中。
- 将 extras 中的 `tomcat-juli.jar` 转换成 `$CATALINA BASE/bin/tomcat-juli.jar`。
- 如果使用安全管理器运行，则需要编辑 `$CATALINA_BASE/conf/catalina.policy` 文件来修改它，以便使用不同版本的 `tomcat-juli.jar`。

```
**注意**：其中的工作原理在于：优先将库加载到      `$CATALINA_HOME` 中同样的库中。

**注意**：`tomcat-juli.jar` 之所以从 `$CATALINA BASE`/bin 加载（而不是从 `$CATALINA_BASE`/lib 加载），是因为它是用作引导进程的，而引导类都是从 bin 加载的。
```

6.删除 `$CATALINA_BASE/conf/logging.properties`，以防止 `java.util.logging` 生成零长度的日志文件。

7.启动 Tomcat。

log4j 配置沿用了默认的 java.util.logging 设置：管理器与主机管理器应用各自获得了独立的日志文件，而所有其余内容都发送到 `catalina.log` 日志文件中。

你可以（也应该）更加挑剔地选择日志所包括的包。Tomcat 使用 Engine 和 Host 名称来定义 logger。比如，要想得到更详细的 Catalina localhost log，可以将它放在 `log4j.properties` 属性中。注意，在 log4j 基于 XML 的配置文件的命名惯例上，目前存在一些问题，所以建议使用所前所述的属性文件，直到未来版本的 log4j 允许使用这种惯例。

```
log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost]=DEBUG
log4j.logger.org.apache.catalina.core=DEBUG
log4j.logger.org.apache.catalina.session=DEBUG
```

警告：设定为 DEBUG 级别，会产生数以兆计的日志，从而拖慢 Tomcat 的启动。只有当需要调试 Tomcat 内部操作，才应该使用这一级别。

你的 Web 应用当然应该使用各自的 log4j 配置。上面的配置是有效的。你可以将相似的 `log4j.properties` 文件放到你的 Web 应用的 `WEB-INF/classes` 目录中，将 log4jx.y.z.jar 放入 `WEB-INF/lib` 中。 然后指定包级别日志。这是基本的 log4j 配置方法，不需要 Commons-Logging。更多选项可参考 log4j 文档，该页面只是一种引导指南。

额外注意：

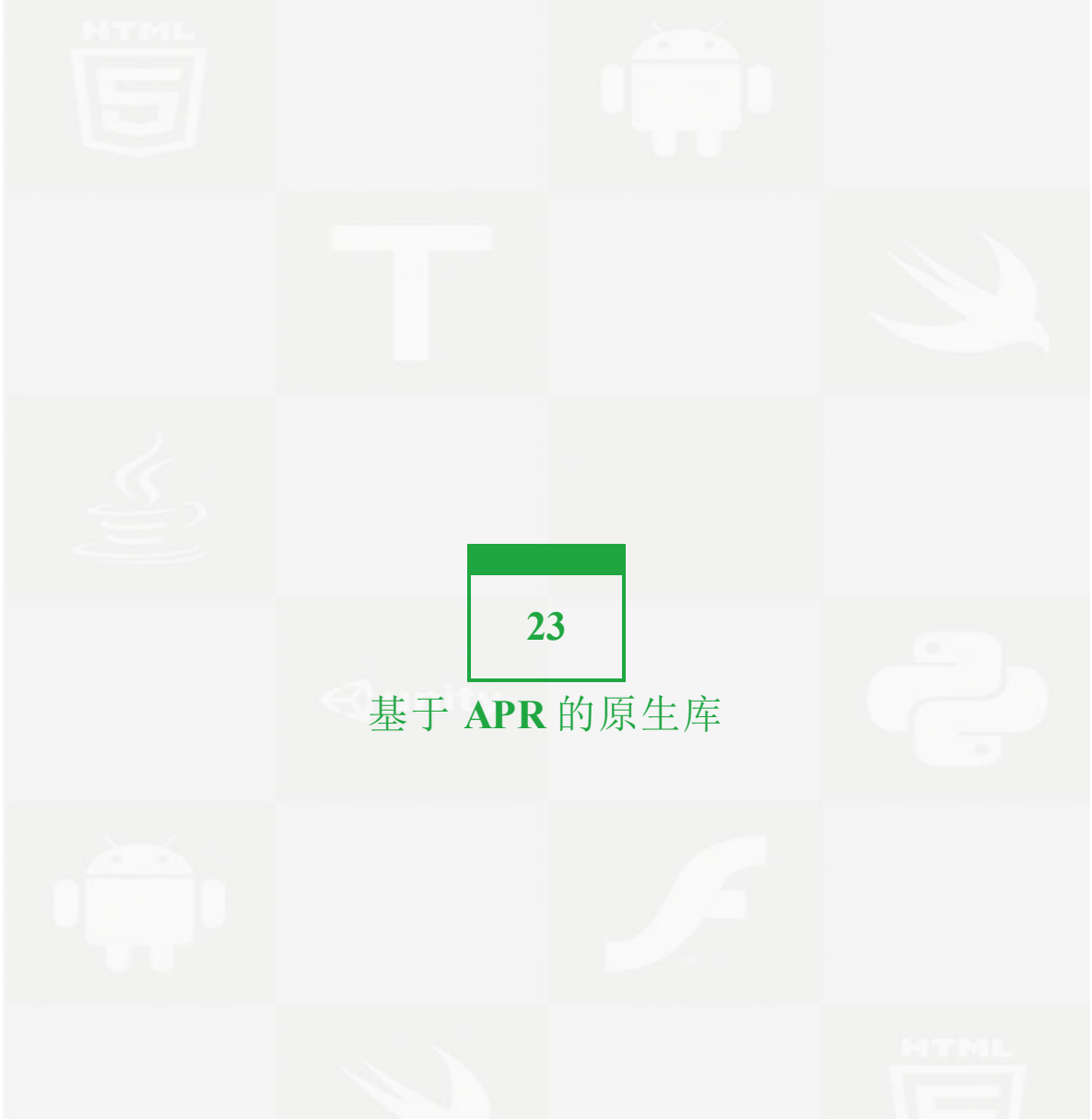
- 通过 Commons 类加载器将 log4j 库暴露给 Web 应用。详见类加载器文档。
  正是由于这一点，使用 [Apache Commons Logging] 库的 Web 应用和库有可能自动会将 log4j 选为底层日志实现。

- `java.util.logging` API 仍适用于直接使用它的 Web 应用。`${catalina.base}/conf/logging.properties` 文件仍然可被 Tomcat 启动脚本所引用。详情可

查看本页的简介部分。

如前面相关步骤所述，删除了 `${catalina.base}/conf/logging.properties` 文件，会导致 `java.util.logging` 回退到 JRE 默认的配置，从而使用 ConsoleHandler，然而却不创建任何标准日志文件。所以必须确保：在禁止标准机制之前，所有的日志文件必须是由 log4j 创建的。

- **Access Log Valve** 和 ** ExtendedAccessLogValve** 使用它们自包含的日志实现，所以无法配置使用 log4j，详情参看 Valves。



23

基于 APR 的原生库

简介

Tomcat 可以使用 Apache Portable Runtime（APR） 来增强可扩展性与性能，并能更好地与原生服务器技术相集成。APR 是一种具有高度可移植性的类库，是 Apache HTTP Server 2.x 的核心。APR 具有许多用途，包括访问高级 IO 功能（比如 sendfile、epoll 和 OpenSSL）、系统级功能（随机数生成、系统状态，等等）以及原生进程处理（共享内存、NT 管道、UNIX 套接字）。

这些特性能让 Tomcat 成为一种通用的 Web 服务器，更使其更好地与原生的 Web 技术相集成。从整体上来说，这使得 Java 越来越有望成为一个成熟的 Web 服务器平台，而不单纯是一种仅仅着重研究后端的技术。

安装

APR 支持需要安装三个关键的原生组件：

- APR 库
- Tomcat 所用的 》JNI 包装器
- OpenSSL 库

Windows

Windows 安装文件从这里下载 32 位或 AMD 64 位，里面是包含 OpenSSL 和 APR 的集合文件。

Linux

多数 Linux 分发版都会自带 APR 与 OpenSSL 包。JNI 包装器（litcnative）然后被编译。它依赖 APR、OpenSSL 与 Java 头。

需要：

- APR 1.2+ 开发头（libarp-1 dev package）
- OpenSSL 》》》
- Java compatible JDK 1.4+》》
- GNU 开发环境（gcc，make）

APR 组件

当所有的库都正确安装好且适用于 Java（如果加载失败，就会显示相关的库路径），Tomcat 连接器就会自动使用 APR。这里，连接器的配置跟通常的配置没什么不同，但会用一些特别的属性来配置 APR 组件。对于大多数用例来说，这些属性的默认值都已经非常适用了，根本不需要再加以微调。

当启用 APR 时，Tomcat 同样也启用了下面这些功能：

- 默认在所有平台安全会话 ID 生成（Linux 之外的平台需要随机数生成使用配置好的熵）。
- 关于Tomcat 进程的内存使用和 CPU 使用情况的 OS 级统计，由status servlet所显示。

配置 APR 生命周期侦听器（APR Lifecycle Listener）

AprLifecycleListener

属性	描述
SSLEngine	所要使用的 SSLEngine 名称。off：不使用 SSL。 引擎。默认值为 on。这将初始化原生的 SSL 引擎， 属性在连接器中》》》。范例： <Listener className="org.apache.catalina.core.Ap SSLEngine="on" /> 请访问 OpenSSL 官方网站以详细了解 SSL 硬件引擎

配置 APR 连接器

HTTP/HTTPS

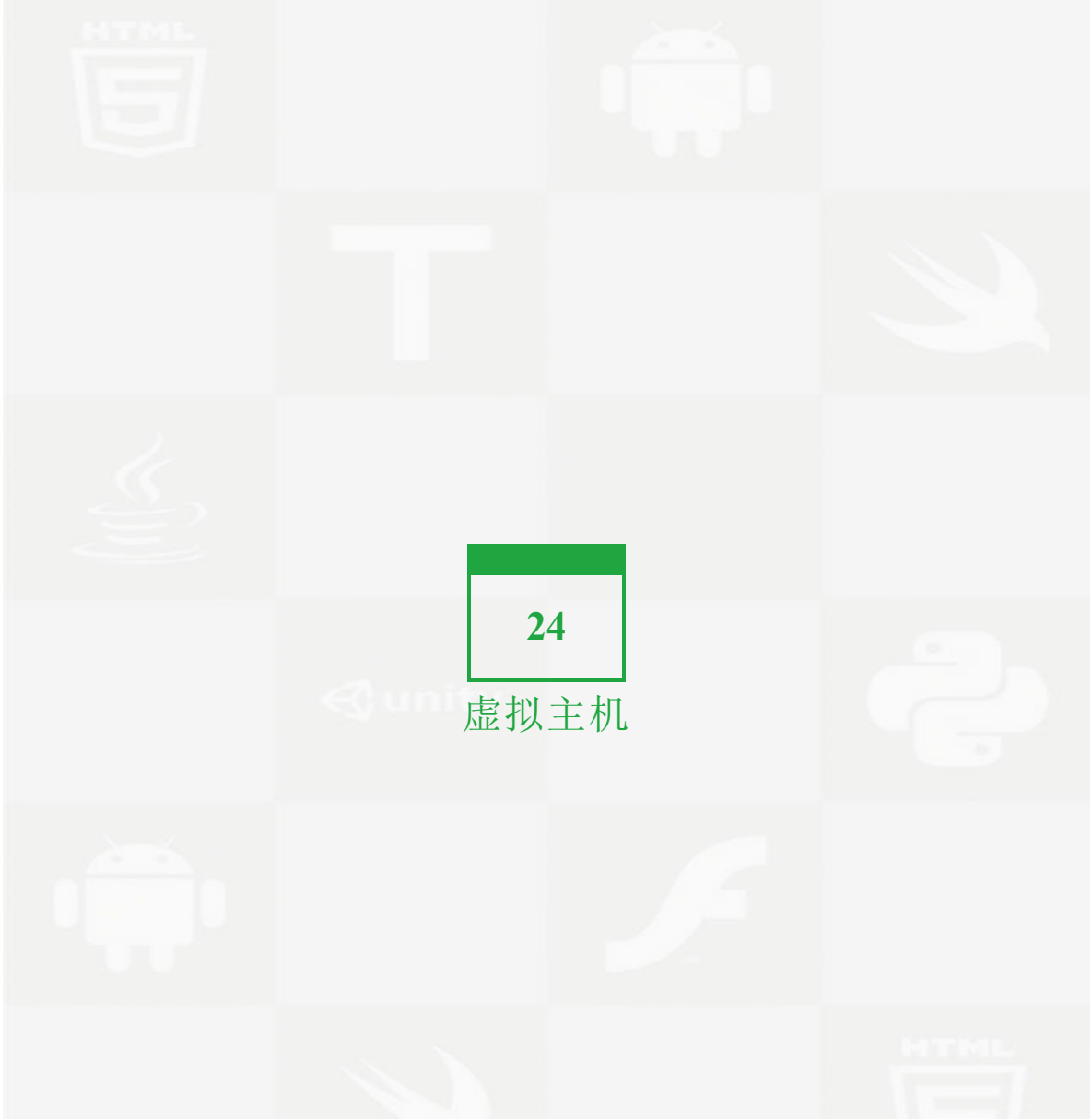
关于 HTTP 配置的相关信息，可查阅 HTTP 连接器配置文档。
关于 HTTPS 配置的相关信息，可查阅 HTTPS 连接器配置文档。

下面这个范例介绍了 SSL 连接器的声明：

```
<Connector port="443" maxHttpHeaderSize="8192"
                 maxThreads="150"
                 enableLookups="false" disableUploadTimeout="true"
                 acceptCount="100" scheme="https" secure="true"
                 SSLEnabled="true"
                 SSLCertificateFile="${catalina.base}/conf/localhost.crt"
                 SSLCertificateKeyFile="${catalina.base}/conf/localhost.key" />
```

AJP

关于 AJP 配置的相关信息，可查阅 AJP 连接器配置文档。



24

虚拟主机

前提设定

针对本教程，假设你有一个开发主机，并有两个主机名：`ren` 和 `stimpy`。再来假设一个 Tomcat 运行实例，`$CATALINA_HOME` 表示它的安装位置，可能是 `/usr/local/tomcat`。

另外，本教程使用 UNIX 风格的分隔符及命令，如果你使用的是 Windows，则需要相应修改一下。

server.xml

编辑 `server.xml` 文件的 Engine 部分，如下所示：

```
<Engine name="Catalina" defaultHost="ren">
    <Host name="ren"    appBase="renapps"/>
    <Host name="stimpy" appBase="stimpyapps"/>
</Engine>
```

注意：每个主机的 appBase 下的目录结构不能彼此重复。

关于 Engine 与 Host 元素的其他属性，可参看相关的配置文档。

Web 应用目录

创建每一个虚拟主机的目录：

```
mkdir $CATALINA_HOME/renapps
mkdir $CATALINA_HOME/stimpyapps
```

配置你的上下文

1. 一般配置方法

上下文通常位于 `appBase` 目录下。比如，在 `ren` 主机上配置 war 文件形式的 `foobar` 上下文，使用 `$CATALINA HOME/renapps/foobar.war`。注意， `ren` 主机的默认或 ROOT 上下文应配置成 `$CATALINA_HOME/renapps/ROOT.war`（WAR 文件形式） 或 `$CATALINA_HOME/renapps/ROOT`（目录形式）。

注意：对于同一主机而言，上下文的 **`docBase`** 不能和 **`appBase`** 相同。

2. context.xml - 方法 1

在上下文中，创建一个 `META-INF` 目录，将你的上下文定义文件（`context.xml`）放入其中，比如说：`$CATALINA_HOME/renapps/ROOT/META-INF/context.xml`。这能使部署更加容易，特别对于分配的是WAR 文件时。

3. context.xml - 方法 2

在 `$CATALINA_HOME/conf/Catalina` 下创建一个结构：

```
mkdir $CATALINA_HOME/conf/Catalina/ren
mkdir $CATALINA_HOME/conf/Catalina/stimpy
```

注意结尾那个名为“Catalina”的目录表示的是如前所示的 Engine 元素的 `name` 属性。

对于默认的 Web 应用，则按如下方式添加：

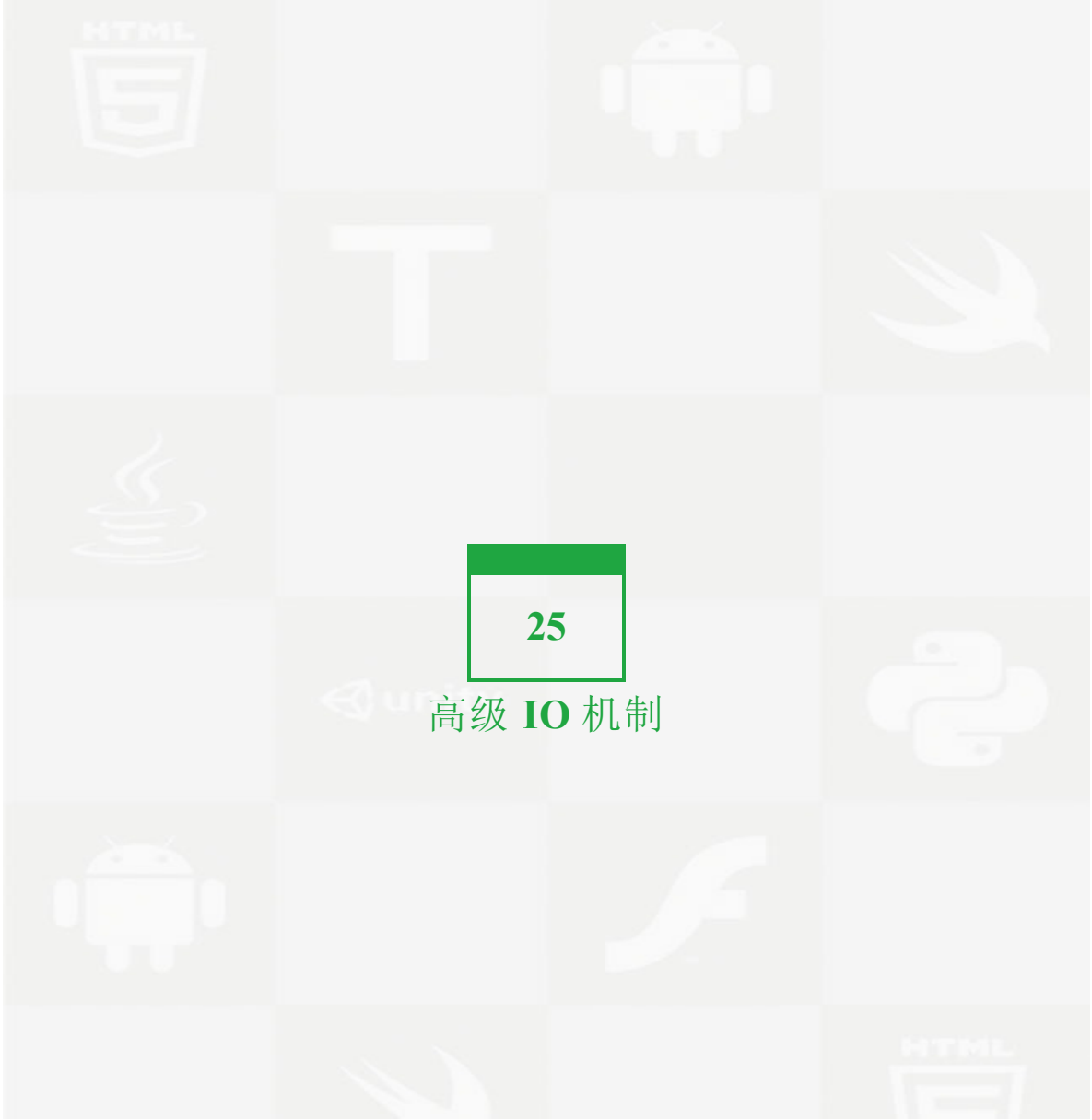
```
$CATALINA_HOME/conf/Catalina/ren/ROOT.xml
$CATALINA_HOME/conf/Catalina/stimpy/ROOT.xml
```

如果想为每个主机都使用 Tomcat Manager 应用，则需要按下列方式来添加它：

```
cd $CATALINA_HOME/conf/Catalina
cp localhost/manager.xml ren/
cp localhost/manager.xml stimpy/
```

4. 更多信息

有关 Context 元素的其他属性，可以参阅相关的配置文档：Context。



25

高级 IO 机制

简介

由于基于 APR 或 NIO API 来构建连接器，Tomcat 能在通常的阻塞 IO 之上提供一些扩展，从而支持 Servlet API。

重要说明：这些特性需要使用 **APR** 或 **NIO HTTP** 连接器。经典的 **java.io HTTP** 连接器 与 **AJP** 连接器并不支持它们。

Comet 支持

Comet 支持能让 Servlet 实现：对 IO 的异步处理；当连接可以读取数据时，接收事件（而不是总使用阻塞读取）；将数据异步地写入连接（很可能是响应其他一些源所产生的事件）。

1. Comet 事件

根据发生的具体事件，实现 `org.apache.catalina.comet.CometProcessor` 接口的 Servlet 将调用自己的事件方法，而非通常的服务方法。事件对象允许访问常见的请求与响应对象，使用方式与通常方式相同。主要的区别在于：在处理 BEGIN 事件到 END 或 ERROR 事件之间，这些事件对象能够保持有效和完整的功能性。事件类型如下：

- `EventType.BEGIN` 在连接处理开始时被调用，用来初始化使用了请求和响应对象的相关字段。从处理完该事件后直到 END 或 ERROR 事件开始处理时的这段时间内，有可能使用响应对象在开放连接中写入数据。注意，响应对象以及所依赖的 OutputStream 和 Writer 仍不能同步，因此在通过多个线程访问它们时，需要进行强制实现同步操作。处理完初始化事件后，就可以提交请求对象了。
- `EventType.READ` 该事件表明可以使用输入数据，读取过程不会阻塞。可以使用 InputStream 或 Reader 的 available 和 ready 方法来确定是否存在阻塞危险：当数据被报告可读时，Servlet 应该进行读取。当读取遇到错误时，Servlet 可以通过正确传播 Exception 属性来报告这一情况。抛出异常会导致 ERROR 事件的调用，连接就会关闭。另外，也有可能捕获一个异常，在 Servlet 可能使用的数据结构上进行清理，然后使用事件的 close 方法。不允许从 Servlet 对象执行方法外部去读取数据。

  在一些平台（比如 Windows）上，利用 READ 事件来表示客户端断开连接。从流中读取的结果可能是 -1、IOException 异常或 EOFException 异常。一定要正确处理这些情况。如果你没有捕捉到 IOException 异常，那么当 Tomcat 捕获到异常时，它会立刻调用你的事件队列生成一个 ERROR 事件来存储这些错误，并且你会马上收到这个消息。
- `EventType.END` 请求处理完毕时，就会调用 END 方法。Begin 方法初始化的字段也将被重置。在处理完这一事件后，请求和响应对象，以及它们所依赖的对象，都将被回收，以便再去处理其他请求。当数据可读取时，以及到达请求输入的文件末尾时（这通常表明客户端通过管线提交请求），也会调用 END。
- `EventType.ERROR`：当连接上出现 IO 异常或类似的不可回收的错误时，容器就会调用 ERROR。在开始时候被初始化的字段在这时候被重置。在处理完这一事件后，请求和响应对象，以及它们所依赖的对象，都将被回收，以便再去处理其他请求。

下面是一些事件子类别，通过它们可以对事件处理过程进行微调（注意：其中有些事件可能需要使用 `org.apache.catalina.valves.CometConnectionManagerValve` 值）：

- `EventSubType.TIMEOUT`: 连接超时（`ERROR` 的子类别）。注意，这个 ERROR 子类型并不是必须的。除非 servlet 使用该事件的 close 方法，否则连接将不会关闭。
- `EventSubType.CLIENT_DISCONNECT`：客户端连接被关闭（`ERROR` 的子类别）。
- `EventSubType.IOEXCEPTION`：表示发生了 IO 异常（比如无效内容），例如无效的块阻塞（`ERROR` 的子类别）。
- `EventSubType.WEBAPP_RELOAD`：重新加载 Web 应用（`END` 的子类别）。
- `EventSubType.SESSION_END`：Servlet 终止了会话（`END` 的子类别）。

如上所述，Comet 请求的典型生命周期会包含一系列的事件：`BEGIN -> READ -> READ -> READ -> ERROR/TIMEOUT`。任何时候，Servlet 都能用事件的 close 方法来终止对请求的处理。

2. Comet 过滤器

跟一般的过滤器一样，当处理 Comet 事件时，就会调用一个过滤器队列。这些过滤器应该实现 `CometFilter` 接口（和常用的过滤器接口一样），在部署描述符文件中的声明与映像也都和通常的过滤器一样。当过滤器队列在处理事件时，它将只含有那些跟所有通常映射规则相匹配的过滤器，并且这些过滤器要实现 CometFilter 接口。

3. 范例代码

在下面的范例伪码中，通过使用上文所述的 API，Servlet 实现了异步聊天功能。

```
public class ChatServlet
    extends HttpServlet implements CometProcessor {

    protected ArrayList<HttpServletResponse> connections =
        new ArrayList<HttpServletResponse>();
    protected MessageSender messageSender = null;

    public void init() throws ServletException {
        messageSender = new MessageSender();
        Thread messageSenderThread =
            new Thread(messageSender, "MessageSender[" +
getServletContext().getContextPath() + "]");
        messageSenderThread.setDaemon(true);
        messageSenderThread.start();
    }

    public void destroy() {
        connections.clear();
        messageSender.stop();
        messageSender = null;
    }

    /**
     * Process the given Comet event.
     *
     * @param event The Comet event that will be processed
     * @throws IOException
     * @throws ServletException
     */
    public void event(CometEvent event)
        throws IOException, ServletException {
        HttpServletRequest request = event.getHttpServletRequest();
        HttpServletResponse response = event.getHttpServletResponse();
        if (event.getEventType() == CometEvent.EventType.BEGIN) {
            log("Begin for session: " + request.getSession(true).getId());
            PrintWriter writer = response.getWriter();
            writer.println("<!DOCTYPE html>");
            writer.println("<head><title>JSP Chat</title></head><body>");
            writer.flush();
            synchronized(connections) {
                connections.add(response);
            }
        } else if (event.getEventType() == CometEvent.EventType.ERROR) {
            log("Error for session: " + request.getSession(true).getId());
            synchronized(connections) {
                connections.remove(response);
            }
            event.close();
        } else if (event.getEventType() == CometEvent.EventType.END) {
            log("End for session: " + request.getSession(true).getId());
```

```
            synchronized(connections) {
                connections.remove(response);
            }
            PrintWriter writer = response.getWriter();
            writer.println("</body></html>");
            event.close();
        } else if (event.getEventType() == CometEvent.EventType.READ) {
            InputStream is = request.getInputStream();
            byte[] buf = new byte[512];
            do {
                int n = is.read(buf); //can throw an IOException
                if (n > 0) {
                    log("Read " + n + " bytes: " + new String(buf, 0, n)
                            + " for session: " + request.getSession(true).getId());
                } else if (n < 0) {
                    error(event, request, response);
                    return;
                }
            } while (is.available() > 0);
        }
    }

    public class MessageSender implements Runnable {

        protected boolean running = true;
        protected ArrayList<String> messages = new ArrayList<String>();

        public MessageSender() {
        }

        public void stop() {
            running = false;
        }

        /**
         * Add message for sending.
         */
        public void send(String user, String message) {
            synchronized (messages) {
                messages.add("[" + user + "]: " + message);
                messages.notify();
            }
        }

        public void run() {

            while (running) {

                if (messages.size() == 0) {
                    try {
                        synchronized (messages) {
                            messages.wait();
                        }
                    } catch (InterruptedException e) {
                        // Ignore
                    }
                }

                synchronized (connections) {
                    String[] pendingMessages = null;
                    synchronized (messages) {
                        pendingMessages = messages.toArray(new String[0]);
```

```
                        messages.clear();
                    }
                    // Send any pending message on all the open connections
                    for (int i = 0; i < connections.size(); i++) {
                        try {
                            PrintWriter writer = connections.get(i).getWriter();
                            for (int j = 0; j < pendingMessages.length; j++) {
                                writer.println(pendingMessages[j] + "<br>");
                            }
                            writer.flush();
                        } catch (IOException e) {
                            log("IOExeption sending message", e);
                        }
                    }
                }

            }

        }

    }

}
```

4. Comet 超时

如果使用 NIO 连接器，你可以为不同的 comet 连接设置单独的超时。只需设置一个如下所示的请求属性即可设置超时：

```
CometEvent event.... event.setTimeout(30*1000);
```

或

```
event.getHttpServletRequest().setAttribute("org.apache.tomcat.comet.timeout", new Integer(30 * 1000));
```

超时被设置为 30 秒。重要说明：为了设置超时，必须完成 BEGIN 事件。默认值为 `soTimeout`。

如果使用 APR 连接器，所有的 Comet 连接将拥有统一的超时值：`soTimeout*50`。

异步写操作

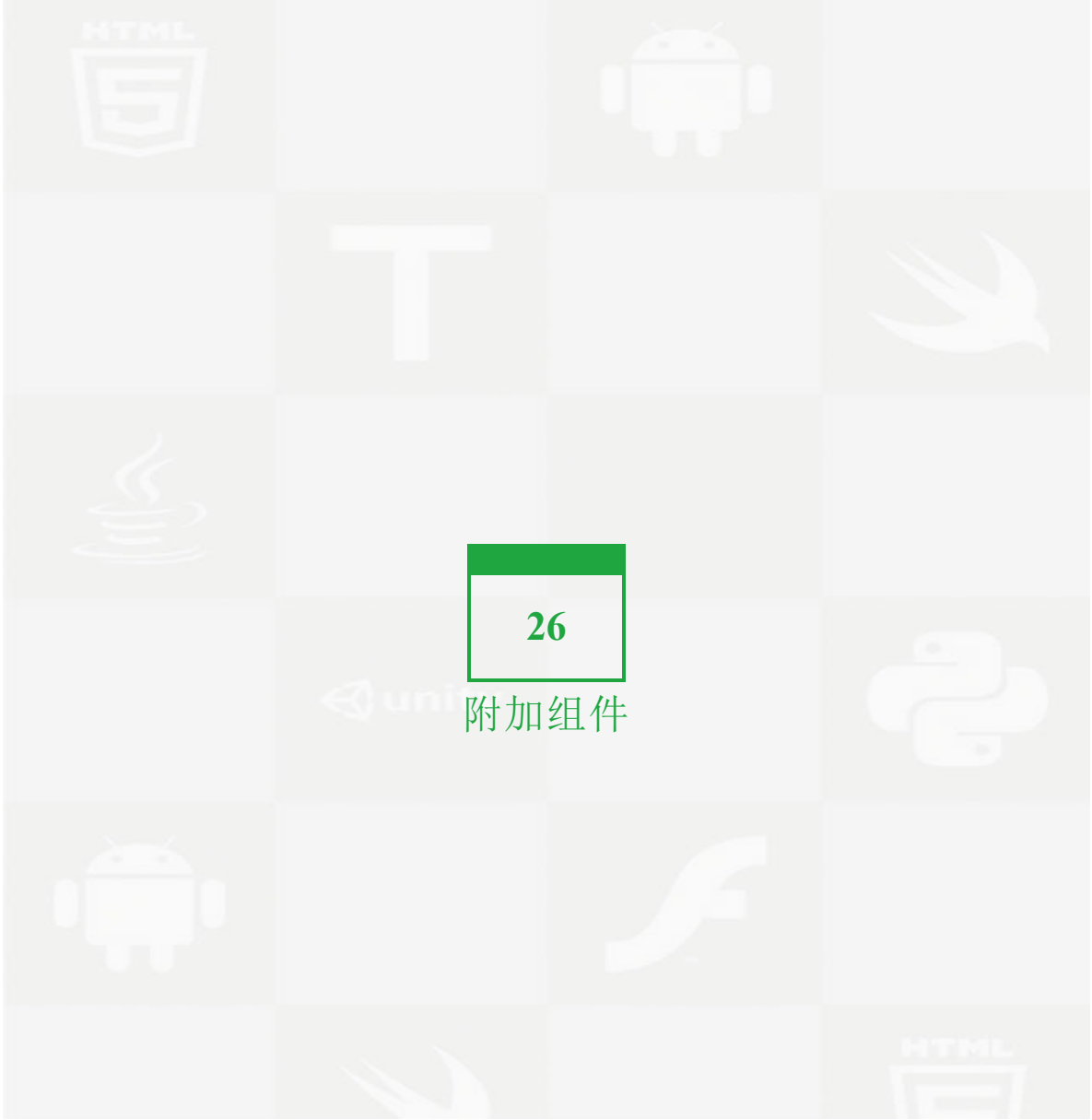
当 APR 或 NIO 可用时，Tomcat 支持使用 sendfile 方式去发送大型静态文件。只要系统负载一增加，就会异步地高效执行写操作。作为一种使用阻塞写操作发送大型响应的替代方式，有可能使用 sendfile 代码来将内容写入静态文件。缓存值将利用这一点将响应数据缓存至文件而非存储在内存中。如果请求属性 `org.apache.tomcat.sendfile.support` 设为 `Boolean.TRUE`，则表示支持 sendfile。

通过合适的请求属性，任何 Servlet 都可以指示 Tomcat 执行 sendfile 调用。正确地设置响应长度也是很有必要的。在使用 sendfile 时，最好确定请求与响应都没有被包装起来。因为稍后连接器本身将发送响应主体，所以不能够过滤响应主体。除了设置 3 个所需的请求属性之外，Servlet 不应该发送任何响应数据，但能使用一些能够修改响应报头的方法（比如设定 cookie）。

- `org.apache.tomcat.sendfile.filename` 作为字符串发送的标准文件名。
- `org.apache.tomcat.sendfile.start` 开始位置偏移值，长整型值。
- `org.apache.tomcat.sendfile.end` 结束位置偏移值，长整型值。

除了设置这些属性，还有必要设置内容长度报头。不要指望 Tomcat 来处理，因为你可能已经将数据写入输出流了。

注意，使用 sendfile 将禁止 Tomcat 可能在响应中执行的压缩操作。



26

附加组件

简介

Tomcat 可以使用许多附件组件。这些附加组件有可能是由用户在需要时创建的，或者是从镜像下载站下载而来的。

下载

打开 Tomcat 下载页面，在“快速导航链接”（Quick Navigation Links）中点击“浏览”（browse）链接。在随后打开页面的 bin/extras 中可以找到附加组件。

构建

附加组件使用 Tomcat 标准的 Ant 脚本的 `extras` 目标构建而成。Ant 脚本位于 Tomcat 的资源包中。

构建过程为：

- 按照 构建指令，从资源包中构建一个 Tomcat 二进制文件（注意：附加组件的构建过程将会用到它，但以后不需要实际用到。）
- 执行命令 `ant extras`，运行构建脚本。
- 附加组件的 JAR 文件放到 `output/extras` 文件夹内。
- 参考下文提到的文档来了解这些 JAR 文件的使用方法。

组件列表

完整的通用日志实现

Tomcat 使用一个改名的包，硬编码的通用日志 API（commons-logging API）实现来使用 java.util.logging API。通用日志额外的组件构建了一个完备的包，重新命名的通用日志实现来替代 Tomcat 所提供的实现。参考日志记录页面了解使用方法。

Web 服务支持（JSR 109）

Tomcat 为可能用于解决 Web 服务引用的 JSR 109 提供了》》工厂。将生成的 catalina-ws.jar 以及 jaxrpc.jar 和 wsdl4j.jar（或 JSR 109 的另一个实现）放在 Tomcat 的 lib 文件夹下。

用户应注意的是，wsdl4j.jar 遵循 CPL 1.0 许可，而不是 Apache License version 2.0。

JMX 远程生命周期侦听器（JMX Remote Lifecycle Listener）

JMX 协议需要 JMX 服务器（在这里指的就是 Tomcat）在两个网络端口上进行侦听。其中一个端口通过配置可以是固定端口，而另外一个则是随机选择的。这就很难穿越防火墙来使用 JMX 。JMX 远端生命周期侦听器能实现两个固定端口，从而简化了穿越防火墙连接到 JMX 的过程。

27

如何在 Maven 中使用 Tomcat 库

Tomcat 快照

Tomcat 快照位于 Apache Snapshot Repository。官方 URL 为：

```
http://people.apache.org/repo/m2-snapshot-repository/org/apache/tomcat/
```

隔段时间就会发布一个版本的快照，没有固定的周期。Tomcat 团队将保证快照改动的有效性。

Tomcat 版本

Tomcat 的稳定版本（Stable release）会发布到 Central Maven Repositories。其 URL 为：

```
http://repo2.maven.org/maven2/org/apache/tomcat/
```

28

安全性注意事项

简介

对于大多数用例来说，默认配置下的 Tomcat 都是相当安全的。有些环境可能需要更多（或更少）的安全配置。本文统一介绍了一下可能影响安全性的配置选项，并适当说明了一下修改这些选项所带来的预期影响。目的是为了在评价 Tomcat 安装时，提供一些应值得考虑的配置选项。

注意：本章内容毕竟有所局限，你还需要对配置文档进行深入研究。在相关文档中有更完整的属性描述。

非 **Tomcat** 设置

Tomcat 配置不应成为唯一的防线，也应该保障系统（操作系统、网络及数据库，等等）中的其他组件的安全。

不应该以根用户的身份来运行 Tomcat，应为 Tomcat 进程创建并分配一个专门的用户，并为该用户配置最少且必要的操作系统权限。比如，不允许使用 Tomcat 用户实现远程登录。

文件权限同样也应适当限制。就拿 ASF 中的 Tomcat 实例为例说明吧（禁止自动部署，Web 应用被部署为扩张的目录。），标准配置规定所有的 Tomcat 文件都由根用户及分组用户所拥有，拥有者具有读写特权，分组只有读取特权，而 World 则没有任何特权。例外之处在于，logs、temp 以及 work 目录的权限都由 Tomcat 用户而不是根用户所拥有。这意味着即使攻击者破坏了 Tomcat 进程，他们也不能改变 Tomcat 配置，无法部署新应用，也无法修改现有应用。Tomcat 进程使用掩码 007 来维护这种权限许可。

对于网络层面，需要使用防火墙来限制进站与出站连接，只允许出现那些你希望的连接。

默认的 Web 应用

概述

Tomcat 安装时自带了一些默认启用的 Web 应用。过去一段时间内发现了不少关于这些应用的漏洞。用不到的应用就该删除，以避免给系统带来相关漏洞而产生的安全风险。

ROOT

ROOT 应用带来安全风险的可能性非常小，但它确实含有正在使用的 Tomcat 的版本号。应该从可公开访问的 Tomcat 实例中清除 ROOT 应用，不是出于安全性原因，而是因为这样能给用户提供一个更适合的默认页面。

Documentation

Documentation 带来安全风险的可能性非常小，但》它标识出了当前正使用的 Tomcat 版本。应该从可公开访问的 Tomcat 实例中清除该应用。

Examples

应该从安全敏感性安装中移除 examples 应用。虽然 examples 应用并不包含任何已知的漏洞，但现已证明，它所包含的一些功能可以被攻击者利用，特别是一些显示所有接收内容，并且能设置新 cookie 的 cookie 范例。攻击者将这些公关和部署在 Tomcat 实例中的另一个应用中的漏洞相结合，就能获取原本根本不可能得到的信息。

Manager

由于 Manager 应用允许远程部署 Web 应用，所以经常被攻击者利用，因为应用的密码普遍强度不够，而且大多在 Manager 应用中启用了 Tomcat 实例可公开访问的功能。Manager 应用默认是不能访问的，因为没有配置能够执行这种访问的用户。如果启用 Manager 应用，就应该遵循 保证管理型应用的安全性 一节中的指导原则。

Host Manager

Host Manager 应用能够创建并管理虚拟主机，包括启用虚拟主机的 Manager 应用。Host Manager 应用默认是不能访问的，因为没有配置能够执行这种访问的用户。如果启用 Host Manager 应用，就应该遵循 保证管理型应用的安全性 一节中的指导原则。

保证管理型应用的安全性

在配置能够为 Tomcat 实例提供管理功能的 Web 应用时，需要遵循下列指导原则：

- 保证任何被允许访问管理应用的用户的密码是强密码。
- 不要放弃使用 LockOutRealm，它能防止暴力破解者攻击用户密码。
- 将 `/META-INF/context.xml` 中的限制访问 localhost 的RemoteAddrValve 取消注释。如果需要远程访问，使用该值可限制到特定的 IP 地址。

安全管理器（Security Manager）

启用安全管理器能让 Web 应用运行在沙盒中，从而极大限制 Web 应用执行恶意行为的能力——比如调用 `System.exit()`，在 Web 应用根目录或临时目录外建立网络连接或访问文件系统。但应注意的是，有些恶意行为是安全管理器所无法阻止的，比如利用无限循环产生 CPU 极大开销。

启用安全管理器经常用来限制潜在影响，比如防止攻击者通过某种方式危害受信任的 Web 应用。安全管理器也可能被用来减少运行不受信任的 Web 应用（比如在托管环境中）所带来的风险，但它只能减少这种风险，并不能终止不受信任的 Web 应用。如果运行多个不受信任的 Web 应用，强烈建议将每个应用都部署为独立的 Tomcat 实例（理想情况下还需要部署在独立的主机上），以便尽量减少恶意 Web 应用对其他应用产生的危害。

Tomcat 已经过安全管理器的测试。但大多数 Tomcat 用户却没有运行过安全管理器，所以 Tomcat 也没有相关的用户测试过的配置。现在已经（并会继续）报告指出了一些关于运行安全管理器时产生的 Bug。

安全管理器在运行时所暴露出的限制可能在于中断很多应用。所以，在未经大量测试前，还是不要使用它为好。理想情况下，安全管理器应该在开发前期使用，因为对于一个成熟的应用来说，启用安全管理器后，记录修补问题会极大浪费时间。

启用安全管理器会改变下列设置的默认值：

- Host 元素的 `deployXML` 属性默认值会被改为 `false`。

server.xml 中的关键配置

1. 综述

默认的 server.xml 包含大量注释，比如一些被注释掉的范例组件定义。去掉这些注释将会使其更容易阅读和理解。

如果某个组件类型没有列出，那么该类型也没有能够直接影响安全的相关设置。

2. server

将 `port` 属性设为 `-1` 能禁用关闭端口。

如果关闭端口未被禁用，会为 **shutdown** 配置一个强密码。

3. 侦听器

如果在 Solaris 上使用 gcc 编译 APR 生命周期侦听器，你会发现 APR 生命周期侦听器并不稳定。如果在 Solaris 上使用 APR（或原生）连接器，需要用 Sun Studio 编译器进行编译。

应该启用并恰当地配置 Security 侦听器。

4. 连接器

默认配置了一个 HTTP 和 AJP 连接器。没有用到的连接器应从 server.xml 中清除掉。

`address` 属性用来控制连接器在哪个 IP 地址上侦听连接。默认，连接器会在所有配置好的 IP 地址上进行侦听。

`allowTrace` 属性可启用能够利于调试的 TRACE 请求。由于一些浏览器处理 TRACE 请求的方式（将浏览器暴露给 XSS 攻击），所以默认是不支持 TRACE 请求的。

`maxPostSize` 属性控制解析参数的 POST 请求的最大尺寸。在整个请求期间，参数会被缓存，所以该值默认会被限制到 2 MB大小，以减少 DOS 攻击的风险。

`maxSavePostSize` 属性控制在 FORM 和 CLIENT-CERT 验证期间，saving of POST requests。在整个验证期间（可能会占用好几分钟），参数会被缓存，所以该值默认会被限制到 4 KB大小，以减少 DOS 攻击的风险。

`maxParameterCount` 属性控制可解析并存入请求的参数与值对（GET + POST）的最大数量。过多的参数将被忽略。如果想拒绝这样的请求，配置FailedRequestFilter。

`xpoweredBy` 属性控制是否 X-Powered-By HTTP 报头会随每一个请求发送。如果发送，则该报头值包含 Servlet 和 JSP 规范版本号、完整的 Tomcat 版本号（比如 Apache Tomcat/8.0）、JVM Vendor 名称，以及 JVM 版本号。默认禁用该报头。该报头可以为合法用户和攻击者提供有用信息。

`server` 属性控制 Server HTTP 报头值。对于 Tomcat 4.1.x 到 8.0.x，该报头默认值为 Apache-Coyote/1.1。该报头为合法用户和攻击者提供的有用信息是有限的。

`SSLEnabled`、`scheme` 和 `secure` 这三个属性可以各自独立设置。这些属性通常应用场景为：当 Tomcat 位于反向代理后面，并且该代理通过 HTTP 或 HTTPS 连接 Tomcat 时。通过这些属性，可查看客户端与代理间（而不是代理与 Tomcat之间）连接的 SSL 属性。例如，客户端可能通过 HTTPS 连接代理，但代理连接 Tomcat 却是通过 HTTP。如果 Tomcat 有必要区分从代理处接收的安全与非安全连接，那么代理就必须使用单独分开的

连接，向 Tomcat 传递安全与非安全请求。如果代理使用 AJP，客户端连接的 SSL 属性会经由 AJP 协议传递，那么就不需要使用单独的连接。

`sslEnabledProtocols` 属性用来确定所使用的 SSL/TLS 协议的版本。从 2014 年发生的 POODLE 攻击起，SSL 协议被认为是不安全的，单独 Tomcat 设置中该属性的安全设置为 `sslEnabledProtocols="TLSv1,TLSv1.1,TLSv1.2"`。

`ciphers` 属性控制 SSL 连接所使用的 cipher。默认使用 JVM 的缺省 cipher。这往往意味着，可用 cipher 列表将包含弱导出级 cipher。安全环境通常需要配置更受限的 cipher 集合。该属性可以利用 OpenSSL 语法格式来包括/排除 cipher 套件。截止 2014 年 11 月 19 日，对于单独 Tomcat 8 与 Java 8，可使用 `sslEnabledProtocols` 属性，并且排除非 DH cipher，以及弱/失效 cipher 来指定 TLS 协议，从而实现正向加密（Forward Secrecy）技术。对于以上这些设定工作来说，Qualys SSL/TLS test是一个非常不错的配置工具。

`tomcatAuthentication` 和 `tomcatAuthorization` 属性都用于 AJP 连接，用于确定 Tomcat 是否应该处理所有的认证和授权，或者是否应委托反向代理来认证（认证用户名作为 AJP 协议的一部分被传递给 Tomcat），而让 Tomcat 继续执行授权。

`allowUnsafeLegacyRenegotiation` 属性提供对 CVE-2009-3555 漏洞（一种 TLS 中间人攻击）的应对方案，应用于 BIO 连接器中。如果底层 SSL 实现易受 CVE-2009-3555 漏洞影响，才有必要使用该属性。参看 Tomcat 8 安全文档可详细了解这种缺陷的当前状态及其可使用的解决方案。

AJP 连接器中的 `requiredSecret` 属性配置了 Tomcat 与 Tomcat 前面的反向代理之间的共享密钥，从而防止通过 AJP 协议进行非授权连接。

5. Host 元素

Host 元素控制着部署。自动部署能让管理更为轻松，但也让攻击者更容易部署恶意应用。自动部署由 `autoDeploy` 和 `deployOnStartup` 属性来控制。如果两个属性值为 `false`，则 server.xml 中定义的上下文将会被部署，任何更改都将需要重启 Tomcat 才能生效。

在 Web 应用不受信任的托管环境中，将 `deployXML` 设置为 `false` 将忽略任何包装 Web 应用的 context.xml，可能会把增加特权赋予 Web 应用。注意，如果启用安全管理器，则 `deployXML` 属性默认为 `false`。

6. Context 元素

`server.xml`、默认的 `context.xml` 文件，每个主机的 `context.xml.default` 文件、Web 应用上下文文件

`crossContext` 属性能够控制是否允许上下文访问其他上下文资源。默认为 `false`，而且只应该针对受信任的 Web 应用。

`privileged` 属性控制是否允许上下文使用容器提供的 servlet，比如 Manager servlet。默认为 `false`，而且只针对受信任的 Web 应用。

内嵌 Resource 元素中的 `allowLinking` 属性控制是否允许上下文使用链接文件。如果启用而且上下文未经部署，那么当删除上下问资源时，也会一并将链接文件删除。默认值为 `false`。在大小写敏感的操作系统上改变该值，将会禁用一些安全措施，并且允许直接访问 WEB-INF 目录。

7. Valve

强烈建议配置 AccessLogValve。默认的 Tomcat 配置包含一个 AccessLogValve。通常会对每个 Host 上进行配置，但必要时也可以在每个 Engine 或 Context 上进行配置。

应通过 RemoteAddrValve 来保护管理应用。注意：这个 Valve 也可以用作过滤器。*allow* 属性用于限制对一些已知信任主机的访问。

默认的 ErrorReportValve 在发送给客户端的响应中包含了 Tomcat 版本号。为了避免这一点，可以在每个 Web 应用上配置自定义错误处理器。另一种方法是，可以配置 `ErrorReportValve`，将其 `showServerInfo` 属性设为 `false`。另外，通过创建带有下列内容的 CATALINA_BASE/lib/org/apache/catalina/util/ServerInfo.properties 文件，可以改变版本号。

```
server.info=Apache Tomcat/8.0.x
```

根据需要来改变该值。注意，这也会改变一些管理工具所报告的版本号，可能难于确定实际安装的版本号。`CATALINA_HOME/bin/version.bat|sh` 脚本依然能够报告版本号。

当出现错误时，默认的 `ErrorReportValve` 能向客户端显示堆栈跟踪信息以及/或者 JSP 源代码。为了避免这一点，可以在每个 Web 应用内配置自定义错误处理器。另一种方法是，可以显式配置一个 `ErrorReportValve`，并将其 `showReport` 属性设为 `false`。

8. Realm

MemoryRealm 并不适用于生产用途，因为要想让 Tomcat-users.xml 中的改动生效，就必须重启 Tomcat。

JDBCRealm 也不建议用于生产环境，因为所有的认证和授权选项都占用一个线程。可以用 DataSourceRealm 来替代它。

UserDatabaseRealm 不适合大规模安装。它适合小规模且相对静态的环境。

JAASRealm 使用并不广泛，因此也不如其他几个 Realm 成熟。在未进行大量测试之前，建议不采用这种 Realm。

默认，Realm 并不能实现账户锁定。这就给暴力破解者带来了方便。要想防范这一点，需要将 Realm 包装在 LockOutRealm 中。

9. Manager

manager 组件用来生成会话 ID。

可以利用 `randomClass` 属性来改变生成随机会话 ID 的类。

可以利用 `sessionIdLength` 属性来改变会话 ID 的长度。

系统属性

将系统属性 `org.apache.catalina.connector.RECYCLE_FACADES` 设为 `true`，将为每一个请求创建一个新的门面（facade）对象，这能减少因为应用 bug 而将一个请求中数据暴露给其他请求的风险。

系统属性 `org.apache.catalina.connector.CoyoteAdapter.ALLOW_BACKSLASH` 和 `org.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH` 允许对请求 URI 的非标准解析。使用这些选项，当处于反向代理后面时，攻击者可以绕过由代理所强制设定的各种安全限制。

如果禁用系统属性 `org.apache.catalina.connector.Response.ENFORCE_ENCODING_IN_GET_WRITER` 可以会带来不利后果。许多违反 RFC2616 的用户代理在应该使用 ISO-8859-1 的规范强制默认值时，试图猜测文本媒体类型的字符编码。一些浏览器会解析为 UTF-7 编码，这样做的后果是：如果某一响应包含的字符对 ISO-8859-1 是安全的，但如果解析为 UTF-7，却能触发 XSS 漏洞。

Web.xml

如果Web 应用中默认的 `conf/web.xml` 和 `WEB-INF/web.xml` 文件定义了本文提到的组件，》》

在配置 DefaultServlet 时，将 `readonly` 设为 `true`。将其变为 `false` 能让客户端删除或修改服务器上的静态资源，进而上传新的资源。由于不需要认证，故而通常也不需要改变。

将 `DefaultServlet` 的 `listings` 设为 `false`。之所以这样设置，不是因为允许目录列表是不安全之举，而是因为要对包含数千个文件的目录生产目录列表，会大量消耗计算资源，会容易导致 DOS 攻击。

`DefaultServlet` 的 `showServerInfo` 设为 `true`。当启用目录列表后，Tomcat 版本号就会包含在发送给客户端的响应中。为了避免这一点，可以将 `DefaultServlet` 的 `showServerInfo` 设为 `false` 。另一种方法是，另外，通过创建带有下列内容的 CATALINA_BASE/lib/org/apache/catalina/util/ServerInfo.properties 文件，可以改变版本号。

```
server.info=Apache Tomcat/8.0.x
```

根据需要来改变该值。注意，这也会改变一些管理工具所报告的版本号，可能难于确定实际安装的版本号。`CATALINA_HOME/bin/version.bat|sh` 脚本依然能够报告版本号。

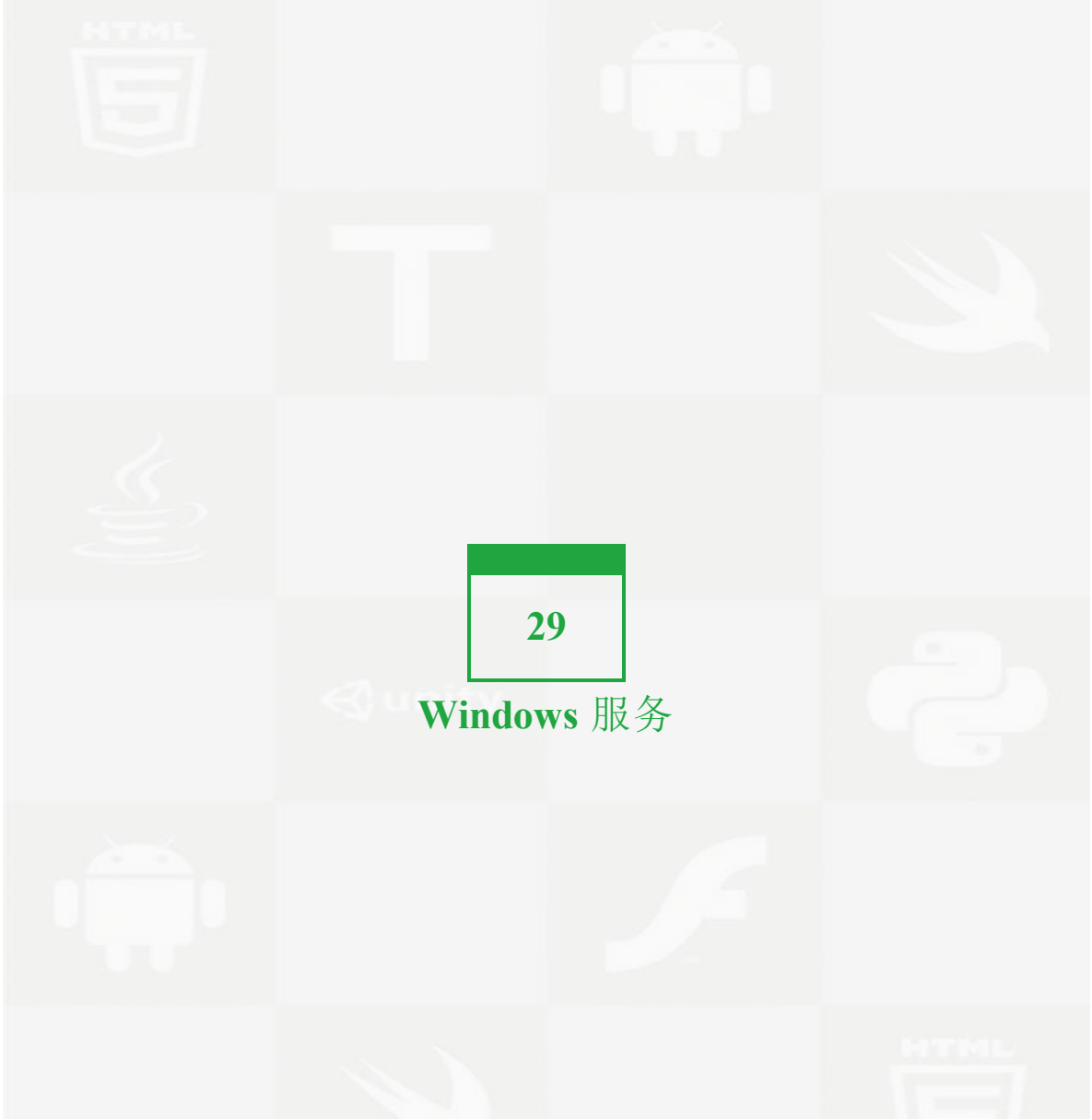
可以设置 `FailedRequestFilter` 来拒绝那些请求参数解析时发生错误的请求。没有过滤器，默认行为是忽略无效或过多的参数。

`HttpHeaderSecurityFilter` 可以为响应添加报头来提高安全性。如果客户端直接访问 Tomcat，你可能就需要启用这个过滤器以及它所设定的所有报头（除非应用已经设置过它们）。如果通过反向代理访问 Tomcat，该过滤器的配置需要与反向代理所设置的任何报头相协调。

总结

BASIC 与 FORM 验证会将用户名及密码存为明文。在不受信任的网络情况下，使用这种认证机制的 Web 应用和客户端间的连接必须使用 SSL。

会话 cookie 加上已认证用户，基本上就将用户密码摆在攻击者面前了，无论何时给予跟密码级别相同的保护。通常这就需要经过 SSL 来认证，或者在整个会话期间都使用 SSL。



29

Windows 服务

Tomcat 服务应用

Tomcat8 是一个服务应用，能使 Tomcat 8 以 Windows 服务的形式运行。

Tomcat 监控应用

Tomcat8w 是一个监控与配置 Tomcat 服务的 GUI 应用。

可用的命令行选项为：

//ES//	编辑服务配置	这是默认操作。如果没有提供其他选项，则调用它。但是可执行未见被重命名为 **servicenamew.exe**。
//MS//	监控服务	将图标放到系统托盘中。

命令行实参

命令行指令格式为：**//XX//ServiceName**。

可用的命令行选项为：

//TS//	以控制台应用的方式运行服务	默认操作。如果没有其他选项，则调用它。ServiceName 是可执行文件没有后缀 exe 的名称，即 Tomcat8。
//RS//	运行服务	只能被服务管理器调用
//SS//	停止服务	
//US//	更新服务参数	
//IS//	安装服务	
//DS//	删除服务	如果服务运行，则停止服务

命令行形参

每一个命令形参都有一个前缀 `--`。如果命令行前缀为 `++`，则该值会附加到已有选项中。如果环境变量和命令行形参相同，但是前缀是 `PR_`，则它要优先处理。比如：

```
set PR_CLASSPATH=xx.jar
```

它等同于把以下作为命令行形参：

```
--Classpath=xx.jar
```

形参名称	默认	描述
`--Description`	-	服务符）
`--DisplayName`	服务名	服务
`--Install`	procrun.exe //RS//ServiceName	安装
`--Starup`	manual	服务**man**
`++DependsOn`	-	该服务。赖服
`++Environment`	-	利用的一字符量。`#` 可须以
`--User`	-	用于户。**java**能使在没的账
`--Password`	-	通过

		户账
`--JavaHome`	JAVA_HOME	设定同的
`--Jvm`	auto	可以 Win JVM 的完境变
`++JvmOptions`	-Xrs	传入为 - 符来用于
`--Classpath`	-	设定 **exe**
`--JvmMs`	-	初始不能
`--JvmMx`	-	内存不能
`--JvmSs`	-	线程能用
`--StartMode`	-	取值之一 • jv 依赖相关 • Ja 自动件。%JA 确保或使确的

		proc 认的 • ex 映像
`--StartImage`	-	运行 **exe**
`--StartPath`	-	start 径
`--StartClass`	Main	包含 **jvm** **exe**
`--StartMethod`	main	方法
`++StartParams`	-	传入 `Sta` 用
`--StopMode`	-	取值 之一 **Star**
`--StopImage`	-	运行 执行
`--StopPath`	-	停止 路径
`--StopClass`	Main	用于 用于
`--StopMethod`	main	方法
`++StopParams`	-	传入 一组 分隔
`--StopTimeout`	没有超时	用于

		结束
`--LogPath`	%SystemRoot%\System32\LogFiles\Apache	定义 建路
`--LogPrefix`	commons-daemon	定义 志文 义的 .YY
`--LogLevel`	Info	定义 些值 一： **Deb**
`--StdOutput`	-	重定 果指 Logl 件名 **stdo** **DAY**
`--StdError`	-	重定 果指 Logl 件名 **stde**
`--PidFile`	-	定义 名。 目录

安装服务

最安全的手动安装服务的方式是利用提供的 **service.bat** 脚本。需要有管理员特权才能运行该脚本。为了安装服务，必要时可以采用 `/user` 指定一个用户。

注意：在 Windows Vista 或其他版本更新的 Windows 操作系统上，如果开启了用户账户控制功能（UAC，User Account Control），当脚本启动 Tomcat8.exe 时，系统会要求提供额外的特权。如果你想为服务安装程序传入附加选项，如 `PR_*` 环境变量，则必须在系统对它们进行全局配置，或者启动相关程序，利用更高级的特权来设置它们，比如：右键点击 cmd.exe 然后选择 “以管理员身份运行”；在 Windows 8（或更新版本）或 Windows Server 2012（或更新版本）系统中，还可以在文件资源管理器中点击“文件”菜单，为当前目录打开一个高级命令提示符（elevated command prompt）。详情参看问题 56143。

```
Install the service named 'Tomcat8'
C:\> service.bat install
```

还有第 2 个可选参数，可以让你指定服务名，就像 Windows 服务所展示的那样。

```
Install the service named 'MyService'
C:\> service.bat install MyService
```

如果使用 tomcat8.exe，你需要使用 **//IS//** 参数。

```
Install the service named 'Tomcat8'
C:\> tomcat8 //IS//Tomcat8 --DisplayName="Apache Tomcat 8" \
C:\> --Install="C:\Program Files\Tomcat\bin\tomcat8.exe" --Jvm=auto \
C:\> --StartMode=jvm --StopMode=jvm \
C:\> --StartClass=org.apache.catalina.startup.Bootstrap --StartParams=start \
C:\> --StopClass=org.apache.catalina.startup.Bootstrap --StopParams=stop
```

更新服务

要想更新服务参数，需要使用 **//US//** 参数。

```
Update the service named 'Tomcat8'
C:\> tomcat8 //US//Tomcat8 --Description="Apache Tomcat Server -
http://tomcat.apache.org/ " \
C:\> --Startup=auto --
Classpath=%JAVA_HOME%\lib\tools.jar;%CATALINA_HOME%\bin\bootstrap.jar
```

如果想为服务指定可选名，需要以如下方式进行：

```
Update the service named 'MyService'
C:\> tomcat8 //US//MyService --Description="Apache Tomcat Server -
http://tomcat.apache.org/ " \
C:\> --Startup=auto --
Classpath=%JAVA_HOME%\lib\tools.jar;%CATALINA_HOME%\bin\bootstrap.jar
```

删除服务

如要删除服务，需使用 **//DS//** 参数。
如果服务正在运行，则会先停止然后再删除。

```
Remove the service named 'Tomcat8'
C:\> tomcat8 //DS//Tomcat8
```

为服务指定可选名的方式如下：

```
Remove the service named 'MyService'
C:\> tomcat8 //DS//MyService
```

调试服务

想要在控制台模式下运行服务，需使用 //**TS**// 参数。通过按下 CTRL+C or CTRL+BREAK 使服务关闭。如果将 tomcat8.exe 重命名为 testservice.exe，那么只需执行 testservice.exe，就会默认执行这个命令模式了。

```
Run the service named 'Tomcat8' in console mode
C:\> tomcat8 //TS//Tomcat8 [additional arguments]
Or simply execute:
C:\> tomcat8
```

多个实例

Tomcat 支持安装多个实例。一个 Tomcat 安装可以带有多个实例，它们可以在不同 IP/端口组合上运行，或者是以多个 Tomcat 版本运行，每个版本都一个或多个实例，在不同的不同 IP/端口组合上运行。

每个实例的文件夹都应具有如下目录结构：

- conf
- logs
- temp
- webapps
- work

conf 目录最起码应该包含 `CATALINA_HOME\conf\` 中下列文件的副本。任何没有复制过或编辑过的文件，将直接从 `CATALINA HOME\conf` 中获取。比如，`CATALINA_BASE\conf` 中的文件就会覆盖 `CATALINA_HOME\conf` 的默认文件。

- server.xml
- web.xml

必须编辑 CATALINA_BASE\conf\server.xml，指定一个唯一的 IP/端口用于实例侦听。找到包含 `<Connector port="8080" ...` 的代码行，添加一个地址属性，并且（或者）更新端口号，以便指定一个唯一的 IP/端口组合。

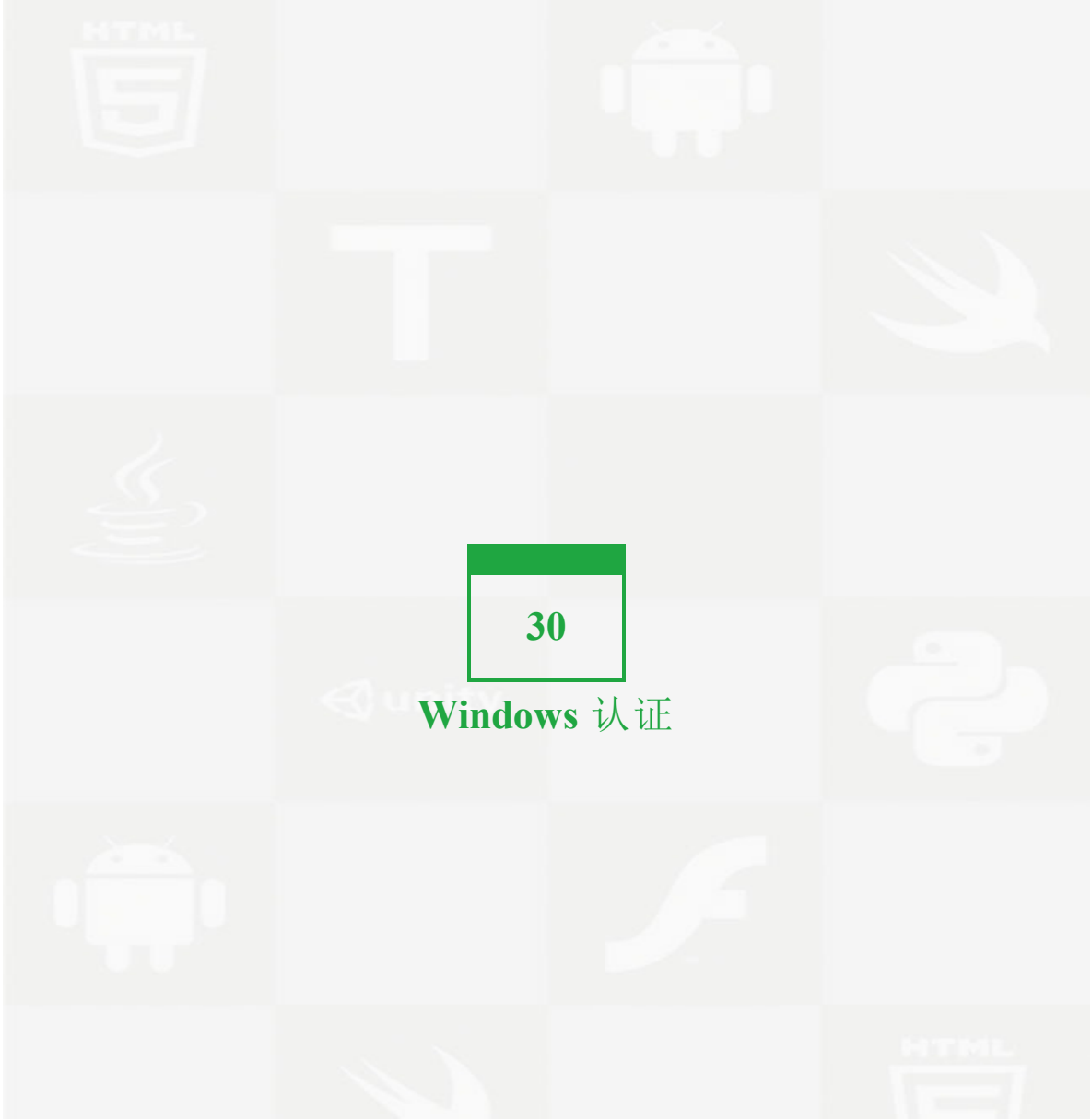
要想安装一个实例，首先将 CATALINA_HOME 环境变量设置为 Tomcat 安装目录名称。然后创建一个第二个环境变量 CATALINA_BASE，并将其指向实例文件夹。最后运行 `service install` 命令指定服务名称。

```
set CATALINA_HOME=c:\tomcat_8
set CATALINA_BASE=c:\tomcat_8\instances\instance1
service install instance1
```

修改服务设置，需要运行 **tomcat8w //ES//instance1**。

对于附加实例，创建附加实例文件夹，更新 `CATALINA_BASE` 环境变量，然后再次安装服务。

```
set CATALINA_BASE=c:\tomcat_8\instances\instance2
service install instance2
```



30

Windows 认证

概述

集成 **Windows** 验证（Integrated Windows authentication）往往用于局域网环境中，因为需要使用服务器执行验证，被验证的用户也必须处于同一域内。为了能够自动验证用户，用户所用的客户端机器也必须处于同一域内。

可以利用以下几种方案来实现 Tomcat 下的集成 Windows 验证：

- 内建 Tomcat 支持。
- 使用第三方库，比如 Waffle。
- 使用支持 Windows 验证的反向代理来执行验证步骤（IIS 或 httpd）。

下面将分别详细讲述这些方案。

内建 Tomcat 支持

需要仔细配置 Kerberos 身份验证服务（集成 Windows 验证的基础）。如果严格按照下列步骤去做，配置就会生效。这些配置的灵活度很小，所以必须严格按照下列方式去做。从测试到现在，已知的规则是：

- 用于访问 Tomcat 服务器的主机名必须匹配 服务主体名称（Service Principal Name，SPN）中的主机名，否则验证就会失败。验证失败时，校验和错误会报告给调试日志。
- 客户端必须明确服务器位于本地可信局域网。
- SPN 必须是 HTTP/<主机名> 的形式，而且必须在所有用到它的位置处保持统一。
- 端口号不能放在 SPN 中。
- 不能将多个 SPN 映射给一个域用户。
- Tomcat 必须以 SPN 关联的域账户或域管理员的身份运行，但不建议采用域管理员的身份运行 Tomcat。
- 在 ktpass 命令中，域名（`DEV.LOCAL`）不区分大小写，在jaas.conf 中也是这样。
- 使用 ktpass 命令时，不能指定域。

在配置 Windows 验证的 Tomcat 内建支持时，共涉及到4个组件：域控制器、托管 Tomcat 的服务器、需要使用 Windows 验证的 Web 应用，以及客户端机器。下面将讲解每个组件所需的配置。

下面配置范例中用到的 3 个机器名称为：`win-dc01.dev.local`（域控制器）、`win-tc01.dev.local`（Tomcat 实例）、`win-pc01.dev.local`（客户端）。它们都是`DEV.LOCAL`域成员。

注意：为了在下面的步骤中使用密码，不得不放宽了域密码规则，对于生产环境，可不建议这么做。

1. 域控制器

下列步骤假设前提是：经过配置，服务器可以做为域控制器来使用。关于如何配置 Windows 服务器配置成域控制器，不在本章讨论范围之内。
配置域控制器，使 Tomcat 支持 Windows 验证的步骤为：

- 创建一个域用户，它将映射到 Tomcat 服务器所用的服务名称上。在本文档中，用户为`tc01`，密码为`tc01pass`。
- 将 SPN 映射到用户账户上。SPN 的形式为：`<service class>/<host>:<port>/<service name>`。本文档所用的 SPN 为`HTTP/win-tc01.dev.local`。要想将用户映射到 SPN 上，运行以下命令：

  ```
  setspn -A HTTP/win-tc01.dev.local tc01
  ```

- 生成 keytab 文件，Tomcat 服务器会用该文件将自身注册到域控制器上。该文件包含用于服务提供者账户的 Tomcat 私钥，所以也应该受到保护。运行以下命令生成该文件（全部命令都应写在同一行中）：

  ```
  ktpass /out c:\tomcat.keytab /mapuser tc01@DEV.LOCAL
        /princ HTTP/win-tc01.dev.local@DEV.LOCAL
        /pass tc01pass /kvno 0
  ```

- 创建客户端所用的域用户。本文档中，域用户为`test`，密码为`testpass`。

以上步骤测试环境为：运行 Windows Server 2008 R2 64 位标准版的域控制器。对于域功能级别和林（forest）功能级别，使用 Windows Server 2003 的功能级别。

2. Tomcat 实例（Windows 服务器）

下列步骤假定前提为：已经安装并配置好了 Tomcat 和 Java 6 JDK/JRE，并以`tc01@DEV.LOCAL`用户来运行

Tomcat。配置用于 Windows 验证的 Tomcat 实例的步骤如下：

- 将域控制器所创建的 `tomcat.keytab` 文件复制到 `$CATALINA_BASE/conf/tomcat.keytab`。
- 创建 kerberos 配置文件 `$CATALINA_BASE/conf/krb5.ini`。本文档使用的文件包含以下内容：

```
    [libdefaults]
    default_realm = DEV.LOCAL
    default_keytab_name = FILE:c:\apache-tomcat-8.0.x\conf\tomcat.keytab
    default_tkt_enctypes = rc4-hmac,aes256-cts-hmac-sha1-96,aes128-cts-    hmac-
sha1-96
    default_tgs_enctypes = rc4-hmac,aes256-cts-hmac-sha1-96,aes128-cts-    hmac-
sha1-96
    forwardable=true

    [realms]
    DEV.LOCAL = {
            kdc = win-dc01.dev.local:88
    }

    [domain_realm]
    dev.local= DEV.LOCAL
    .dev.local= DEV.LOCAL
```

该文件的位置可以通过 `java.security.krb5.conf` 系统属性来修改。

- 创建 JAAS 逻辑配置文件 `$CATALINA_BASE/conf/jaas.conf`。本文档使用的文件包含以下内容：

```
    com.sun.security.jgss.krb5.initiate {
    com.sun.security.auth.module.Krb5LoginModule required
    doNotPrompt=true
    principal="HTTP/win-tc01.dev.local@DEV.LOCAL"
    useKeyTab=true
    keyTab="c:/apache-tomcat-8.0.x/conf/tomcat.keytab"
    storeKey=true;
};

    com.sun.security.jgss.krb5.accept {
        com.sun.security.auth.module.Krb5LoginModule required
        doNotPrompt=true
        principal="HTTP/win-tc01.dev.local@DEV.LOCAL"
        useKeyTab=true
        keyTab="c:/apache-tomcat-8.0.x/conf/tomcat.keytab"
        storeKey=true;
    };
```

本文件位置可以通过 `java.security.auth.login.config` 系统属性来修改。所用的 LoginModule 是 JVM 所专有的，从而能保证所指定的 LoginModule 匹配所用的 JVM。登录配置名称必须与验证 valve 所用值相匹配。

SPNEGO 验证器适用于任何 Realm，但如果和 JNDI Realm 一起使用的话，JNDI Realm 默认将使用用户的委托凭证（delegated credentials）连接 Active 目录。

上述步骤测试环境为：Tomcat 服务器运行于 Windows Server 2008 R2 64 位标准版上，带有 Oracle 1.6.0_24 64 位 JDK。

3. Tomcat 实例（Linux 服务器）

测试环境如下：

- Java 1.7.0, update 45, 64-bit
- Ubuntu Server 12.04.3 LTS 64-bit

- Tomcat 8.0.x (r1546570)

虽然建议使用最新的稳定版本，但其实所有 Tomcat 8 的版本都能使用。

配置与 Windows 基本相同，但存在以下一些差别：

- Linux 服务器不必位于 Windows 域。
- 应该更新 krb5.ini 和 jass.conf 中的 keytab 文件路径，以便适应使用 Linux 文件路径风格（比如：/usr/local/tomcat/...）的 Linux 服务器。

4. Web 应用

配置 Web 应用，以便使用 web.xml 中的 Tomcat 专有验证方法 `SPNEGO`（而不是 `BASIC` 等）。和其他的验证器一样，通过显式地配置验证 valve并且在 Valve 中设置属性来自定义行为。

5. 客户端

配置客户端，以便使用 Kerberos 认证。对于 IE 浏览器来说，这就需要 Tomcat 实例位于“本地局域网”安全域中，并且需要在“工具 > Internet 选项 > 高级”中启用集成 Windows 认证。注意：客户端和 Tomcat 实例不能使用同一台机器，因为 IE 会使用未经证实的 NTLM 协议。

6. 参考资料

正确配置 Kerberos 验证是有一定技巧性的。下列参考资料有一定帮助。一般来说，Tomcat 用户邮件列表中的建议也是可取的。

1. IIS 与 Kerberos
2. SourceForge 的 SPNEGO 项目
3. Oracle Java GSS-API 教程（Java 7）
4. Oracle Java GSS-API 教程 - 疑难解答（Java 7）
5. 用于 Windows 验证的Geronimo 配置
6. Kerberos 交换中的加密选择
7. 受支持的 Kerberos Cipher 套件

第三方库

1. Waffle

关于该解决方案的完整详情，可浏览 Waffle 网站。其关键特性为：

- Drop-in
- 配置简单（无需 JAAS 或 keytab 配置）
- 使用原生库

2. Spring Security - Kerberos 扩展

关于该解决方案的完整详情，可浏览 Kerberos 扩展网站。其关键特性为：

- xx
- 需要生成 Kerberos keytab 文件
- 纯粹 Java 解决方案

3. SourceForge 的 SPNEGO 项目

关于该解决方案的完整详情，可浏览 该项目网站。其关键特性为：

- 使用 Kerberos。
- 纯 Java 解决方案。

4. Jespa

关于该解决方案的完整详情，可浏览 该项目网站。其关键特性为：

- 纯 Java 解决方案
- 高级 Active 目录集成

反向代理

1. Microsoft IIS

通过配置 IIS 提供 Windows 验证的步骤如下：

1. 将 IIS 配置成 Tomcat 的反向代理（参看 IIS Web 服务器文档）。
2. 配置 IIS 使用 Windows 验证。
3. 将 AJP 连接器上的 `tomcatAuthentication` 属性设为 `false`，从而配置 Tomcat 使用来自 IIS 的验证用户信息。另一种方法是，将 `tomcatAuthorization` 设为 `true`，从而在Tomcat 执行授权时，允许 IIS 进行验证。

2. Apache httpd

Apache httpd 默认并不支持 Windows 验证，但可以使用很多第三方模块来实现：

1. 针对 Windows 平台的 mod_auth_sspi
2. 针对非 Windows 平台的 mod_auth_ntlm_winbind。目前已知适用于 32 位平台上的 httpd 2.0.x。有些用户已经报告了 httpd 2.2.x 构建与 64 位Linux 构建所存在的稳定性问题。

采用以下步骤配置 httpd，以便提供 Windows 验证：

1. 将 httpd 配置成 Tomcat 的反向代理（参看 Apache httpd Web 服务器文档）。
2. 配置 httpd 使用 Windows 验证。
3. 将 AJP 连接器上的 `tomcatAuthentication` 属性设为 `false`，从而配置 Tomcat 使用来自 httpd 的验证用户信息。

31

Tomcat 的 JDBC 连接池

简介

JDBC 连接池 `org.apache.tomcat.jdbc.pool` 是 Apache Commons DBCP 连接池的一种替换或备选方案。

那究竟为何需要一个新的连接池？

原因如下：

1. Commons DBCP 1.x 是单线程。为了线程安全，在对象分配或对象返回的短期内，Commons 锁定了全部池。但注意这并不适用于 Commons DBCP 2.x。
2. Commons DBCP 1.x 可能会变得很慢。当逻辑 CPU 数目增长，或者试图借出或归还对象的并发线程增加时，性能就会受到影响。高并发系统受到的影响会更为显著。注意这并不适用于 Commons DBCP 2.x。
3. Commons DBCP 拥有 60 多个类。tomcat-jdbc-pool 核心只有 8 个类。因此为了未来需求变更着想，肯定需要更少的改动。我们真正需要的只是连接池本身，其余的只是附属。
4. Commons DBCP 使用静态接口，因此对于指定版本的 JRE，只能采用正确版本的 DBCP，否则就会出现 `NoSuchMethodException` 异常。
5. 当DBCP 可以用其他更简便的实现来替代时，实在不值得重写那 60 个类。
6. Tomcat JDBC 连接池无需为库本身添加额外线程，就能获取异步获取连接。
7. Tomcat JDBC 连接池是 Tomcat 的一个模块，依靠 Tomcat JULI 这个简化了的日志架构。
8. 使用 `javax.sql.PooledConnection` 接口获取底层连接。
9. 防止饥饿。如果池变空，线程将等待一个连接。当连接返回时，池就将唤醒正确的等待线程。大多数连接池只会一直维持饥饿状态。

Tomcat JDBC 连接池还具有一些其他连接池实现所没有的特点：

1. 支持高并发环境与多核/CPU 系统。
2. 接口的动态实现。支持 java.sql 与 java.sql 接口（只要 JDBC 驱动），甚至在利用低版本的 JDK 来编译时。
3. 验证间隔时间。我们不必每次使用单个连接时都进行验证，可以在借出或归还连接时进行验证，只要不低于我们所设定的间隔时间就行。
4. 只执行一次查询。当与数据库建立起连接时，只执行一次的可配置查询。这项功能对会话设置非常有用，因为你可能会想在连接建立的整个时段内都保持会话。
5. 能够配置自定义拦截器。通过自定义拦截器来增强功能。可以使用拦截器来采集查询统计，缓存会话状态，重新连接之前失败的连接，重新查询，缓存查询结果，等等。由于可以使用大量的选项，所以这种自定义拦截器也是没有限制的，跟 `java.sql/javax.sql` 接口的 JDK 版本没有任何关系。
6. 高性能。后文将举例展示一些性能差异。
7. 极其简单。它的实现非常简单，代码行数与源文件都非常少，这都有赖于从一开始研发它时，就把简洁当做重中之重。对比一下 c3p0，它的源文件超过了 200 个（最近一次统计），而 Tomcat JDBC 核心只有 8 个文件，连接池本身则大约只有这个数目的一半，所以能够轻易地跟踪和修改可能出现的 Bug。
8. 异步连接获取。可将连接请求队列化，系统返回 `Future<Connection>`。
9. 更好地处理空闲连接。不再简单粗暴地直接把空闲连接关闭，而是仍然把连接保留在池中，通过更为巧妙的算法控制空闲连接池的规模。
10. 可以控制连接应被废弃的时间：当池满了即废弃，或者指定一个池使用容差值，发生超时就进行废弃处理。
11. 通过查询或语句来重置废弃连接计时器。允许一个使用了很长时间的连接不因为超时而被废弃。这一点是通过使用 `ResetAbandonedTimer` 来实现的。
12. 经过指定时间后，关闭连接。与返回池的时间相类似。
13. 当连接要被释放时，获取 JMX 通知并记录所有日志。它类似于 `removeAbandonedTimeout`，但却不需要采取任何行为，只需要报告信息即可。通过 `suspectTimeout` 属性来实现。
14. 可以通过 `java.sql.Driver`、`javax.sql.DataSource` 或 `javax.sql.XADataSource` 获取连接。

通过 `dataSource` 与 `dataSourceJNDI` 属性实现这一点。

15. 支持 XA 连接。

使用方法

对于熟悉 Commons DBCP 的人来说，转而使用 Tomcat 连接池是非常简单的事。从其他连接池转换过来也非常容易。

1. 附加功能

除了其他多数连接池能够提供的功能外，Tomcat 连接池还提供了一些附加功能：

- `initSQL` 当连接创建后，能够执行一个 SQL 语句（只执行一次）。
- `validationInterval` 恰当地在连接上运行验证，同时又能避免太多频繁地执行验证。
- `jdbcInterceptors` 灵活并且可插拔的拦截器，能够对池进行各种自定义，执行各种查询，处理结果集。下文将予以详述。
- `fairQueue` 将 fair 标志设为 true，以达成线程公平性，或使用异步连接获取。

2. Apache Tomcat 容器内部

在Tomcat JDBC 文档中，Tomcat 连接池被配置为一个资源。唯一的区别在于，你必须指定 `factory` 属性，并将其值设为 `org.apache.tomcat.jdbc.pool.DataSourceFactory`。

3. 独立性

连接池只有一个从属文件，tomcat-juli.jar。要想在使用 bean 实例化的单一项目中使用池，实例化的 Bean 为 `org.apache.tomcat.jdbc.pool.DataSource`。下文讲到将连接池配置为 JNDI 资源时会涉及到同一属性，也是用来将数据源配置成 bean 的。

4. JMX

连接池对象暴露了一个可以被注册的 MBean。为了让连接池对象创建 MBean，`jmxEnabled` 标志必须设为 true。这并不是说连接池会注册到 MBean 服务器。在像 Tomcat 这样的容器中，Tomcat 本身注册就在 MBean 服务器上注册了 DataSource。`org.apache.tomcat.jdbc.pool.DataSource` 对象会注册实际的连接池 MBean。如果你在容器外运行，可以将 DataSource 注册在任何你指定的对象名下，然后将这种注册传播到底层池。要想这样做，你必须调用 `mBeanServer.registerMBean(dataSource.getPool().getJmxPool(),objectname)`。在调用之前，一定要保证通过调用 `dataSource.createPool()` 创建了池。

属性

为了能够顺畅地在 Commons DBCP 与 Tomcat JDBC 连接池 之间转换，大多数属性名称及其含义都是相同的。

1. JNDI 工厂与类型

属性	描述
`factory`	必需的属性，其值应为 `org.apache.tomcat.jdbc.pool.DataSourceF`
`type`	类型应为 `javax.sql.DataSource` 或 `javax.sql.XADataSource`。 根据类型，将创建 `org.apache.tomcat.jdbc.pool.DataSource` 或 `org.apache.tomcat.jdbc.pool.XADataSourc`

2. 系统属性

系统属性作用于 JVM 范围，影响创建于 JVM 内的所有池。

属性
`org.apache.tomcat.jdbc.pool.onlyAttemptCurrentCla`

3. 常用属性

属性	描述
`defaultAutoCommit`	（布尔值）连接置，则默认采用 `setAutoComm`
`defaultReadOnly`	（布尔值）连接不会调用 `setR` 式，比如：infor
`defaultTransactionIsolation`	（字符串）连接为：（参考 java

	• NONE • READ COMMI • READ UNCOM • REPEATABLE • SERIALIZAB 如果未设置该值
defaultCatalog	（字符串）连接
driverClassName	（字符串）所要动必须能从与 to
username	（字符串）传入意，DataSour方法默认不会使息。可参看 alt
password	（字符串）传入意，DataSour方法默认不会使息。可参看 alt
maxActive	（整形值）池同 100 。
maxIdle	（整型值）池始 maxActive:1 能），留滞时间闲连接将会被释
minIdle	（整型值）池始败，则连接池会参考 testWhi
initialSize	（整型值）连接
maxWait	（整型值）在抛回连接的最长时
testOnBorrow	（布尔值）默认

	进行验证。如果 一个。注意：为 必须为非空字符 validationI
testOnReturn	（布尔值）默认 行验证。注意： 数必须为非空字
testWhileIdle	（布尔值）是否 象。如果对象验 true 值生效， 该属性默认值为 置该值。（另请
validationQuery	（字符串）在将 的 SQL 查询。如 是不抛出 SQLE 为：SELECT 1 dual（Oracle）
validationQueryTimeout	（整型值）连接 validationQ java.sql.St 实现。池本身并 若该值小于或等
validatorClassName	（字符串）实现 org.apache. 了一个无参（可 将通过该类来创 查询。默认为 n 为：com.myco
timeBetweenEvictionRunsMillis	（整型值）空闲 秒计）。不能低 接的频率，以及
numTestsPerEvictionRun	（整型值）Tomc
minEvictableIdleTimeMillis	（整型值）在被

	最短时间（以毫
accessToUnderlyingConnectionAllowed	（布尔值）没有 unwrap 来访问 介绍，或者通过 为 javax.sql
removeAbandoned	（布尔值）该值 removeAband 被设置为 true removeAband 应予以清除。若 恢复该应用的数 为 false 。
removeAbandonedTimeout	（整型值）在废 数。默认为 60 时间最长的查询
logAbandoned	（布尔值）标志 录。由于生成堆 取连接的开销。
connectionProperties	（字符串）在建 符串格式必须为 password 属性会 null。
poolPreparedStatements	（布尔值）未使
maxOpenPreparedStatements	（整型值）未使

4. Tomcat JDBC 增强属性

属性	描述
initSQL	字符串值。当连接第一次
jdbcInterceptors	字符串。继承自类 org.

	子类类名列表，由分号分 器。 这些拦截器将会插入到 预定义的拦截器有： • org.apache.tomca • ConnectionState- 态。 • org.apache.tomca • StatementFinaliz 有关更多预定义拦截器的
validationInterval	长整型值。为避免过度验 行验证。如果连接应该进 对其进行验证。默认为
jmxEnabled	布尔值。是否利用 JMX
fairQueue	布尔值。假如想用真正的 true 。对空闲连接列表 org.apache.tomcat true 。如果想使用异步 设置该标志可保证线程能 在性能测试时，锁及锁等 根据所运行的操作系统， 性 os.name = linux 列，那么只需在连接池类 org.apache.tomcat 添加到系统属性上。
abandonWhenPercentageFull	整型值。除非使用中连接 的百分比，否则不会关闭 默认值为 0，意味着只要
maxAge	长整型值。连接保持时间 否达到 now - time-w 则关闭该连接，不再将其

	在将连接返回池中时，不
useEquals	布尔值。如果想让 Prox 为 true；若想在对比方 用于任何已添加的拦截器 true。
suspectTimeout	整型值。超时时间（以秒 类似于 removeAbandoc 关闭连接。如果 logAb 于或等于 0，则不会执行 弃，或者废弃检查被禁用 记录下 WARN 信息并发送
rollbackOnReturn	布尔值。如果 autoCom 调用回滚方法，从而终止
commitOnReturn	布尔值。如果 autoCom 调用提交方法，从而完成 属性。默认值为 false
alternateUsernameAllowed	布尔值。出于性能考虑， DataSource.getCon 化的具有全局配置属性 但经过配置，连接池还可 项在 DataSource.get 能，只需将 alternate 如果你请求一个连接，凭 2/password 2 凭证，那么 式，池的容量始终以全局 默认值为 false。 该属性作为一个改进方案
dataSource	（javax.sql.DataSource） 而不是利用 java.sql. 非连接字符串）来池化文
dataSourceJNDI	字符串。在 JNDI 中查找 看 datasource 属性的

`useDisposableConnectionFacade`	布尔值。如果希望在连接 则要将值设为 `true`。这 接上查询。默认值为 `tr`
`logValidationErrors`	布尔值。设为 `true` 时， 为 SEVERE。考虑到了向
`propagateInterruptState`	布尔值。传播已中断的线 兼容性，默认值为 `fals`
`ignoreExceptionOnPreLoad`	布尔值。在初始化池时， 为 `false` 时，抛出异常

高级用法

1. JDBC 拦截器

要想看看拦截器使用方法的具体范例，可以看看 `org.apache.tomcat.jdbc.pool.interceptor.ConnectionState`。这个简单的拦截器缓存了三个属性：`autoCommit`、`readOnly`、`transactionIsolation`，为的是避免系统与数据库之间无用的往返。

当需求增加时，姜维连接池核心增加更多的拦截器。欢迎贡献你的才智！

拦截器当然并不局限于 `java.sql.Connection`，当然也可以对方法调用的任何结果进行包装。你可以构建查询性能分析器，以便当查询运行时间超过预期时间时提供 JMX 通知。

2. 配置 JDBC 拦截器

JDBC 拦截器是通过 **jdbcInterceptor** 属性来配置的。该属性值包含一列由分号分隔的类名。如果这些类名非完全限定，就会在它们的前面加上 `org.apache.tomcat.jdbc.pool.interceptor.` 前缀。

范例：
```
jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
```
它实际上等同于：
```
jdbcInterceptors="ConnectionState;StatementFinalizer"
```

拦截器也同样有属性。拦截器的属性指定在类名后的括号里，如果设置多个属性，则用逗号分隔开。

范例：

```
jdbcInterceptors="ConnectionState;StatementFinalizer(useEquals=true)"
```

系统会自动忽略属性名称、属性值以及类名前后多余的空格字符。

org.apache.tomcat.jdbc.pool.JdbcInterceptor

所有拦截器的抽象基类，无法实例化。

属性	描述
useEquals	（布尔值）如果希望 `ProxyConnection` 类使用 `String.equals`，则设为 true；当希望在对比方法名时使用 `==`，则设为 `false`。默认为 `true`。

org.apache.tomcat.jdbc.pool.interceptor.ConnectionState

它能为下列属性缓存连接：`autoCommit`、`readOnly`、`transactionIsolation` 及 `catalog`。这是一种性能增强功能，当利用已设定的值来调用 getter 与 setter 时，它能够避免往返数据库。

org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer

跟踪所有使用 `createStatement`、`prepareStatement` 或 `prepareCall` 的语句，当连接返回池后，关闭这些语句。

属性	描述
trace	（以字符串形式表示的布尔值）对未关闭语句进行跟踪。当启用跟踪且连接被关闭时，如果相关语句没有关闭，则拦截器会记录所有的堆栈跟踪。默认值为 `false`。

org.apache.tomcat.jdbc.pool.interceptor.StatementCache

缓存连接中的 `PreparedStatement` 或 `CallableStatement` 实例。

它会针对每个连接对这些语句进行缓存，然后计算池中所有连接的整体缓存数，如果缓存数超过了限制 `max`，就不再对随后的语句进行缓存，而是直接关闭它们。

属性	描述
`prepared`	（以字符串形式表示的布尔值）对使用 `prepareStatement` 调用创建的 `PreparedStatement` 实例进行缓存。默认为 `true`
`callable`	（以字符串形式表示的布尔值）对使用 `prepareCall` 调用创建的 `CallableStatement` 实例进行缓存。默认为 `false`
`max`	（以字符串形式表示的整型值）连接池中的缓存语句的数量限制。默认为 `50`

org.apache.tomcat.jdbc.pool.interceptor.StatementDecoratorInterceptor

请参看 48392。拦截器会包装语句和结果集，从而防止对使用了 `ResultSet.getStatement().getConnection()` 和 `Statement.getConnection()` 方法的实际连接进行访问。

org.apache.tomcat.jdbc.pool.interceptor.QueryTimeoutInterceptor

当新语句创建时，自动调用 `java.sql.Statement.setQueryTimeout(seconds)`。池本身并不会让查询超时，完全是依靠 JDBC 驱动来强制查询超时。

属性	描述
`queryTimeout`	（以字符串形式表示的整型值）查询超时的毫秒数。默认为 `1000` 毫秒。

org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReport

当查询超过失败容差值时，记录查询性能并发布日志项目。使用的日志级别为 `WARN`。

属性	描述
`threshold`	（以字符串形式表示的整型值）查询应超时多少毫秒才发布日志警告。默认为 `1000` 毫秒
`maxQueries`	（以字符串形式表示的整型值）为保留内存空间，所能记录的最大查询数量。默认为 `1000`
`logSlow`	（以字符串形式表示的布尔值）如果想记录较慢的查询，设为 `true`。默认为 `true`
`logFailed`	（以字符串形式表示的布尔值）如果想记录失败查询，设为 `true`。默认为 `true`

org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReportJmx

这是对 `SlowQueryReport` 的扩展，除了发布日志项目外，它还发布 JMX 通知，以便监视工具作出相关反应。该类从其父类继承了所有属性。它使用了 Tomcat 的 JMX 引擎，所以在 Tomcat 容器外部是无效的。使用该类时，默认情况下，是通过 `ConnectionPool` MBean 来发送 JMX 通知。如果 `notifyPool=false`，则 `SlowQueryReportJmx` 也可以注册一个 MBean。

属性	描述
`notifyPool`	（以字符串形式表示的布尔值）如果希望用 `Slow` 为 `true`
`objectName`	字符串。定义一个有效的 `javax.management.O` 务器上。默认值为 `null`。可以使用 `tomcat.jdbc:type=org.apache.tomcat.j`

`name-of-the-pool` 来注册对象。

org.apache.tomcat.jdbc.pool.interceptor.ResetAbandonedTimer

当连接签出池中后，废弃计时器即开始计时。这意味着如果超时为 30 秒，而你使用连接运行了 10 个 10秒的查询，那么它就会被标为废弃，并可能依靠 `abandonWhenPercentageFull` 属性重新声明。每次成功地在连接上执行操作或执行查询时，该拦截器就会重设签出计时器。

代码范例

其他 JDBC 用途的 Tomcat 配置范例可以参考 相关的 Tomcat 文档。

简单的 Java

下面这个简单的范例展示了如何创建并使用数据源：

```
  import java.sql.Connection;
  import java.sql.ResultSet;
  import java.sql.Statement;

  import org.apache.tomcat.jdbc.pool.DataSource;
  import org.apache.tomcat.jdbc.pool.PoolProperties;

  public class SimplePOJOExample {

      public static void main(String[] args) throws Exception {
          PoolProperties p = new PoolProperties();
          p.setUrl("jdbc:mysql://localhost:3306/mysql");
          p.setDriverClassName("com.mysql.jdbc.Driver");
          p.setUsername("root");
          p.setPassword("password");
          p.setJmxEnabled(true);
          p.setTestWhileIdle(false);
          p.setTestOnBorrow(true);
          p.setValidationQuery("SELECT 1");
          p.setTestOnReturn(false);
          p.setValidationInterval(30000);
          p.setTimeBetweenEvictionRunsMillis(30000);
          p.setMaxActive(100);
          p.setInitialSize(10);
          p.setMaxWait(10000);
          p.setRemoveAbandonedTimeout(60);
          p.setMinEvictableIdleTimeMillis(30000);
          p.setMinIdle(10);
          p.setLogAbandoned(true);
          p.setRemoveAbandoned(true);
          p.setJdbcInterceptors(
            "org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;"+
            "org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer");
          DataSource datasource = new DataSource();
          datasource.setPoolProperties(p);

          Connection con = null;
          try {
            con = datasource.getConnection();
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery("select * from user");
            int cnt = 1;
            while (rs.next()) {
                System.out.println((cnt++)+". Host:" +rs.getString("Host")+
                  " User:"+rs.getString("User")+"
Password:"+rs.getString("Password"));
            }
            rs.close();
            st.close();
```

```
            } finally {
              if (con!=null) try {con.close();}catch (Exception ignore) {}
            }
        }

    }
```

作为资源使用

下例展示了如何为 JNDI 查找配置资源。

```
<Resource name="jdbc/TestDB"
          auth="Container"
          type="javax.sql.DataSource"
          factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
          testWhileIdle="true"
          testOnBorrow="true"
          testOnReturn="false"
          validationQuery="SELECT 1"
          validationInterval="30000"
          timeBetweenEvictionRunsMillis="30000"
          maxActive="100"
          minIdle="10"
          maxWait="10000"
          initialSize="10"
          removeAbandonedTimeout="60"
          removeAbandoned="true"
          logAbandoned="true"
          minEvictableIdleTimeMillis="30000"
          jmxEnabled="true"

jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
            org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
          username="root"
          password="password"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/mysql"/>
```

异步连接获取

Tomcat JDBC 连接池支持异步连接获取，无需为池库添加任何额外线程。这是通过在数据源上添加一个方法 `Future<Connection> getConnectionAsync()` 来实现的。为了使用异步获取，必须满足两个条件：

1. 必须把 `failQueue` 属性设为 `true`。
2. 必须把数据源转换为 `org.apache.tomcat.jdbc.pool.DataSource`。

下例就使用了异步获取功能：

```
  Connection con = null;
  try {
    Future<Connection> future = datasource.getConnectionAsync();
    while (!future.isDone()) {
      System.out.println("Connection is not yet available. Do some background
work");
      try {
        Thread.sleep(100); //simulate work
      }catch (InterruptedException x) {
```

```
        Thread.currentThread().interrupt();
      }
    }
    con = future.get(); //should return instantly
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("select * from user");
```

拦截器

对于启用、禁止或修改特定连接或其组件的功能而言，使用拦截器无疑是一种非常强大的方式。**There are many different use cases for when interceptors are useful**。默认情况下，基于性能方面的考虑，连接池是无状态的。连接池本身所插入的状态是 `defaultAutoCommit`、`defaultReadOnly`、`defaultTransactionIsolation`，或 `defaultCatalog`（如果设置了这些状态）。这 4 个状态只有在连接创建时才设置。无论这些属性是否在连接使用期间被修改，池本身都不能重置它们。

拦截器必须扩展自 `org.apache.tomcat.jdbc.pool.JdbcInterceptor` 类。该类相当简单，你必须利用一个无参数构造函数。

```
  public JdbcInterceptor() {
  }
```

当从连接池借出一个连接时，拦截器能够通过实现以下方法，初始化这一事件或以一些其他形式来响应该事件。

```
public abstract void reset(ConnectionPool parent, PooledConnection con);
```

上面这个方法有两个参数，一个是连接池本身的引用 `ConnectionPool parent`，一个是底层连接的引用 `PooledConnection con`。

当调用 `java.sql.Connection` 对象上的方法时，会导致以下方法被调用：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

`Method method` 是被调用的实际方法，`Object[] args` 是参数。通过观察下面这个非常简单的例子，我们可以解释如果当连接已经关闭时，如何让 `java.sql.Connection.close()` 的调用变得无用。

```
  if (CLOSE_VAL==method.getName()) {
      if (isClosed()) return null; //noop for already closed.
  }
  return super.invoke(proxy,method,args);
```

池启动与停止

当连接池开启或关闭时，你可以得到相关通知。可能每个拦截器类只通知一次，即使它是一个实例方法。也可能使用当前未连接到池中的拦截器来通知你。

```
  public void poolStarted(ConnectionPool pool) {
  }

  public void poolClosed(ConnectionPool pool) {
  }
```

当重写这些方法时，如果你扩展自 `JdbcInterceptor` 之外的类，不要忘记调用超类。

配置拦截器

拦截器可以通过 `jdbcInterceptors` 属性或 `setJdbcInterceptors` 方法来配置。拦截器也可以有属性，可以通过如下方式来配置：

```
  String jdbcInterceptors=

"org.apache.tomcat.jdbc.pool.interceptor.ConnectionState(useEquals=true,fast=yes)"
```

拦截器属性

既然拦截器也有属性，那么你也可以读取其中的属性值。你可以重写 `setProperties` 方法。

```
  public void setProperties(Map<String, InterceptorProperty> properties) {
     super.setProperties(properties);
     final String myprop = "myprop";
     InterceptorProperty p1 = properties.get(myprop);
     if (p1!=null) {
         setMyprop(Long.parseLong(p1.getValue()));
     }
  }
```

获取实际的 JDBC 连接

连接池围绕实际的连接创建包装器，为的是能够正确地池化。同样，为了执行特定的功能，我们也可以在这些包装器中创建拦截器。如果不需要获取实际的连接，可以使用 `javax.sql.PooledConnection` 接口。

```
  Connection con = datasource.getConnection();
  Connection actual = ((javax.sql.PooledConnection)con).getConnection();
```

构建

下面利用 1.6 来构建 JDBC 连接池代码，但它也可以向后兼容到 1.5 运行时环境。为了单元测试，使用 1.6 或更高版本。

更多的关于 JDBC 用途的 Tomcat 配置范例可参看 Tomcat 文档。

从源代码构建

构建非常简单。池依赖于 `tomcat-juli.jar`，在这种情况下，需要 `SlowQueryReportJmx`。

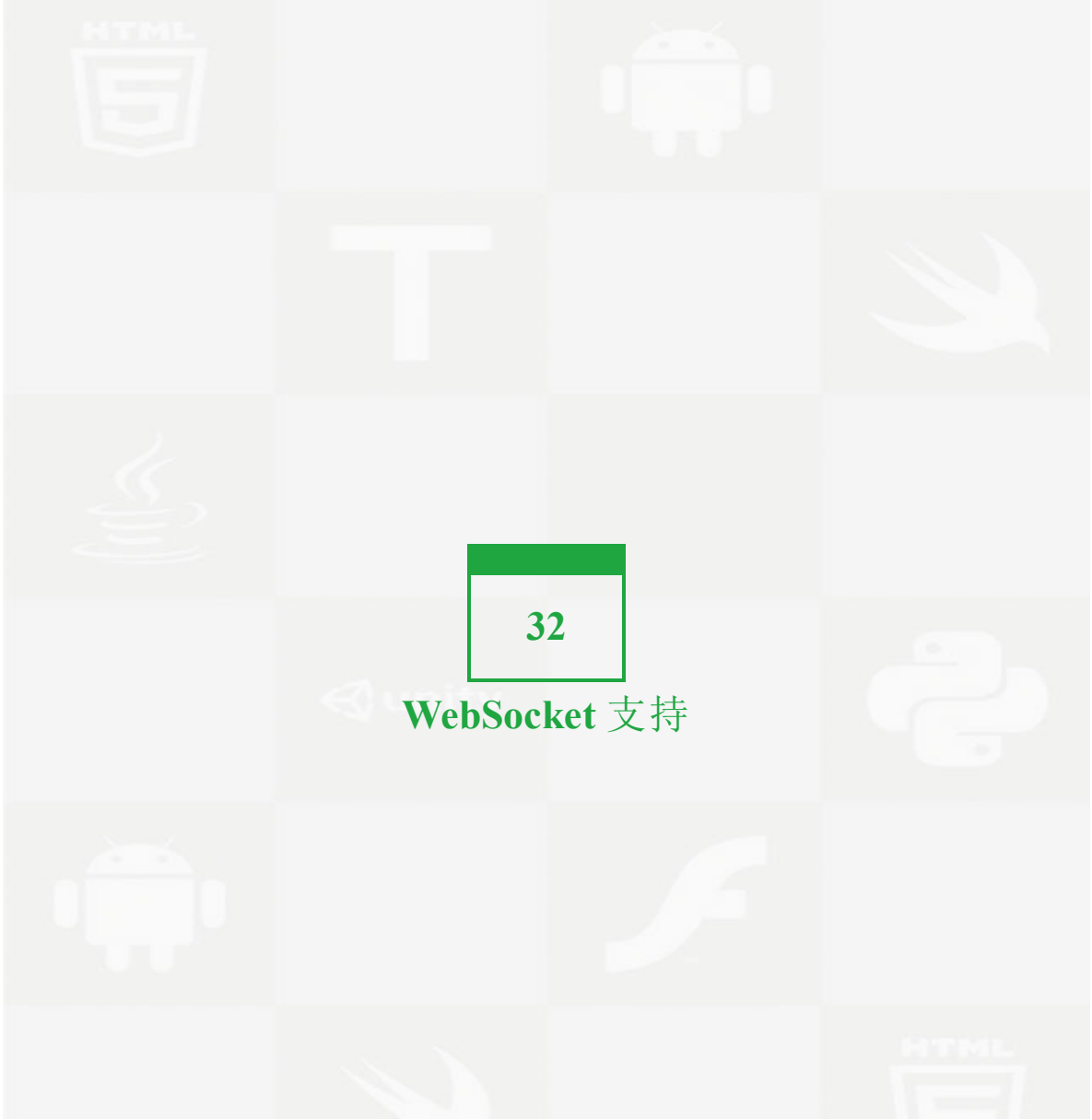
```
javac -classpath tomcat-juli.jar \
      -d . \
      org/apache/tomcat/jdbc/pool/*.java \
      org/apache/tomcat/jdbc/pool/interceptor/*.java \
      org/apache/tomcat/jdbc/pool/jmx/*.java
```

构建文件位于 Tomcat 的源代码仓库中。

为了方便起见，在通过简单构建命令生成所需文件的地方也包含了一个构建文件。

```
ant download  (downloads dependencies)
ant build     (compiles and generates .jar files)
ant dist      (creates a release package)
ant test      (runs tests, expects a test database to be setup)
```

系统针对 Maven 构建进行组织，但是没有生成发布组件，只有库本身。



32

WebSocket 支持

简介

Tomcat 支持由 RFC 6455 所定义的 WebSocket。

应用开发

Tomcat 实现由 JSR-356 定义的 Java WebSocket 1.1 API。

关于 WebSocket API 的使用方法，可查看相关范例，既需要查看客户端 HTML 代码，也需要查看服务器代码。

生产用途

虽然 WebSocket 实现可以和任何 HTTP 连接器一起使用，但并不建议和 BIO HTTP 连接器一起使用，因为 WebSocket 典型用途（大量连接很多时候都是空闲的）并不是很适合 HTTP BIO 连接器，因为该连接器需要不管连接是否空闲，每个连接都应该分配一个线程。

目前，已有报告（56304）发现，Linux 会用大量时间来报告删除的连接。当利用 BIO HTTP 连接器使用 WebSocket 时，当在这段时间内写入时，就容易产生线程阻塞。这似乎不是一种理想的解决方案。使用内核网络参数 `/proc/sys/net/ipv4/tcp_retries2`，可以减少报道删除的连接所花费的时间。或者可以选择另一种 HTTP 连接器，因为它们使用的是非阻塞 IO，从而能让 Tomcat 实现自己的超时机制来解决这些问题。

Tomcat WebSocket 特定配置

Tomcat 为 WebSocket 提供了一些 Tomcat 专有配置选项。这些配置将来有望能进入 WebSocket 正式规范中。

以阻塞模式发送 WebSocket 消息所用的写入超时默认值为 20000 毫秒（20 秒）。通过设定连接到 WebSocket 会话的用户属性集合中的 `org.apache.tomcat.websocket.BLOCKING_SEND_TIMEOUT` 属性，我们可以修改这个写入超时属性值。该属性值类型应该为 `Long`，以毫秒表示所用的超时时间，`-1` 表示超时无限。

如果应用没有为传入的二进制消息定义 `MessageHandler.Partial`，那么必须先对任何传入的二进制消息进行缓存，继而可以通过调用一个已注册专用于处理二进制消息的 `MessageHandler.Whole` 来传递整个消息。默认用于二进制消息的缓存容量是 8192 字节。在应用中，将servlet 上下文初始参数 `org.apache.tomcat.websocket.binaryBufferSize` 设置为期望的字节值，就能修改这个缓存容量。

如果应用没有为传入的文本消息定义 `MessageHandler.Partial`，那么必须先对任何传入的文本消息进行缓存，继而可以通过调用一个已注册专用于处理文本消息的 `MessageHandler.Whole` 来传递整个消息。默认用于文本消息的缓存容量是 8192 字节。在应用中，将servlet 上下文初始参数 `org.apache.tomcat.websocket.textBufferSize` 设置为期望的字节值，就能修改这个缓存容量。

Java WebSocket 规范 1.0 并不允许第一个服务端点开始 WebSocket 握手之后进行程序性部署。默认情况下，Tomcat 继续允许额外的程序性部署。这一行为是通过 servlet 上下文初始化参数 `org.apache.tomcat.websocket.noAddAfterHandshake` 来控制的。将系统属性 `org.apache.tomcat.websocket.STRICT_SPEC_COMPLIANCE` 变为 `true`，可以改变默认值，但是 servlet 上下文中的显式设置却总是优先的。

Java WebSocket 规范 1.0 要求，异步写操作的回调所在的线程应不同于初始化写操作的线程。因为容器线程池并未通过 Servlet API 暴露出来，所以 WebSocket 实现必须提供自己的线程池。这种线程池通过下列 Servlet 上下文初始化参数来控制：

- `org.apache.tomcat.websocket.executorCoreSize`：executor 线程池的核心容量大小。如果不设置，则采用默认值 0。注意，executor 线程池最大允许容量被硬编码至 `Integer.MAX_VALUE`，实际上可以认为该值是无限的。
- `org.apache.tomcat.websocket.executorKeepAliveTimeSeconds`：在终止前，空闲线程在 executor 线程池中留存的最长时间。如未指定，则采用默认的 60 秒。

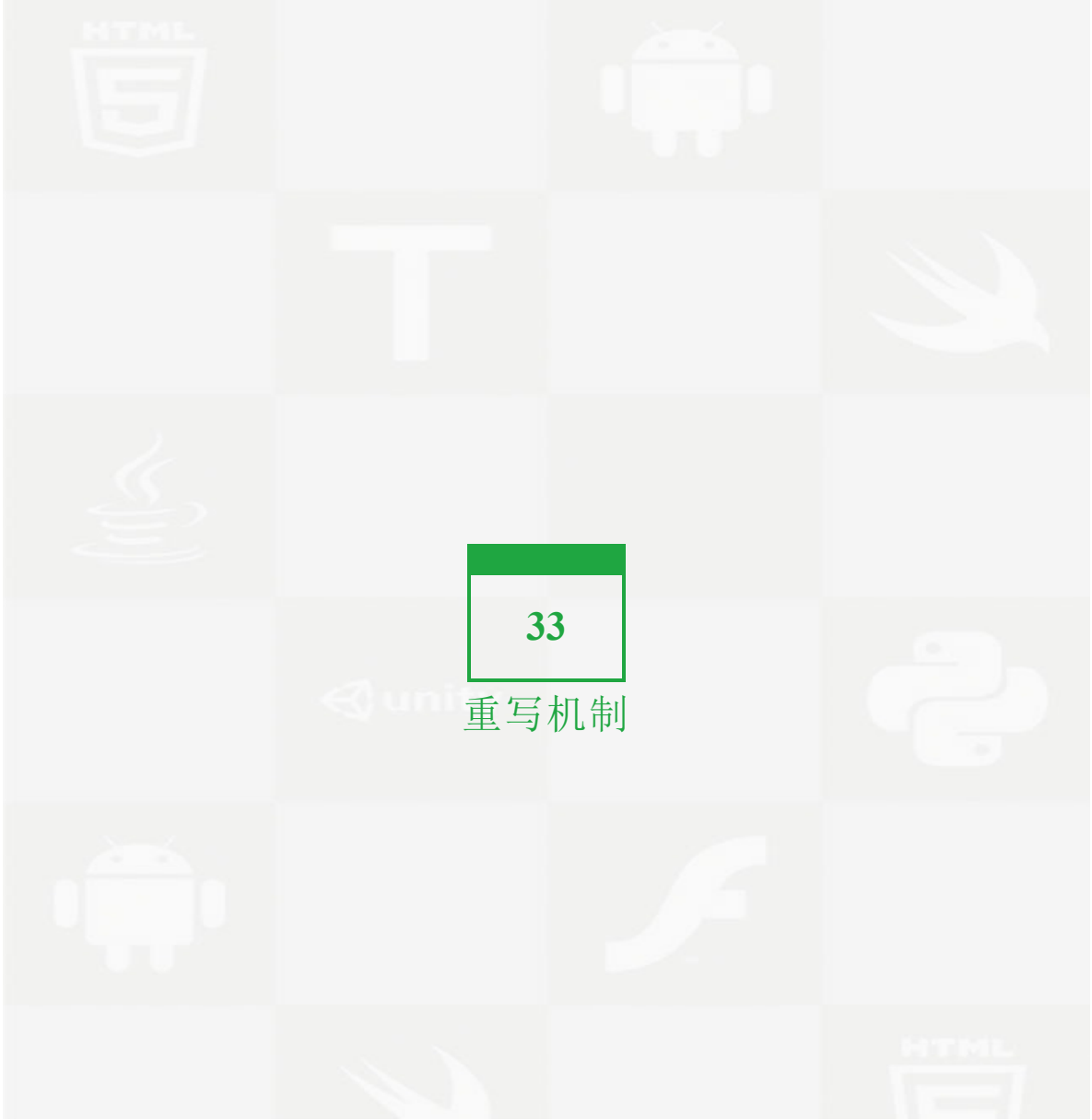
在使用 WebSocket 客户端来连接服务端点时，建立该连接的 IO 超时是通过提供的 `javax.websocket.ClientEndpointConfig` 的 `userProperties` 来控制的。该属性是 `org.apache.tomcat.websocket.IO_TIMEOUT_MS`，是以字符串形式表示的超时时间（以毫秒计），默认为 5000（5 秒）。

在使用 WebSocket 客户端来连接安全的服务端点时，客户端 SSL 配置是通过提供的 `javax.websocket.ClientEndpointConfig` 的 `userProperties` 来控制的。提供以下用户属性：

- `org.apache.tomcat.websocket.SSL_CONTEXT`
- `org.apache.tomcat.websocket.SSL_PROTOCOLS`
- `org.apache.tomcat.websocket.SSL_TRUSTSTORE`
- `org.apache.tomcat.websocket.SSL_TRUSTSTORE_PWD`

默认的信任存储密码（truststore password）为：`changeit`。

如果设置了 `org.apache.tomcat.websocket.SSL_CONTEXT` 属性，则将忽略这两个属性：`org.apache.tomcat.websocket.SSL_TRUSTSTORE` 和 `org.apache.tomcat.websocket.SSL_TRUSTSTORE_PWD`。



33 重写机制

简介

重写 **Valve**（Rewrite Valve） 实现 URL 重写功能的方式非常类似于 Apache HTTP Server 的 `mod_rewrite` 模块。

配置

重写 Valve 是通过使用 `org.apache.catalina.valves.rewrite.RewriteValve` 类名来配置成 Valve 的。

经过配置，重写 Valve 可以做为一个 Valve 添加到 Host 中。参考虚拟服务器文档来了解配置详情。该 Valve 使用包含重写指令的 `rewrite.config` 文件，且必须放在 Host 配置文件夹中。

另外，重写 valve 也可以用在 Web 应用的 context.xml 中。该 Valve 使用包含重写指令的 `rewrite.config` 文件，且必须放在 Web 应用的 `WEB-INF` 文件夹中。

指令

`rewrite.config` 文件包含一系列指令，这些指令和 `mod_rewrite` 所用的指令很像，尤其是核心的 `RewriteRule` 与 `RewriteCond` 指令。

注意：该部分内容修改自 `mod_rewrite` 文档，后者版权归属于 Apache 软件基金会（1995-2006），遵循 Apache 许可发布。

1. RewriteCond

格式：`RewriteCond TestString CondPattern`

`RewriteCond` 指令定义了一个规则条件。一个或多个 `RewriteCond` 指令可以优先于 `RewriteRule` 指令执行。如果 URI 当前状态匹配它的模式，并且满足了这些条件，才会使用下列规则。

TestString 是一种字符串，除了简单的文本之外，它还可以含有下列扩展结构。

- `RewriteRule backreferences` 对形式 `$N`（0 <= N <= 9）的反向引用。提供对模式成组部分的访问（括号中的），从属于 `RewriteCond` 条件当前状态的 `RewriteRule`。
- `RewriteCond backreferences`
- `RewriteMap expansions`
- `Server-Variables` 这些是形式 `%{ NAME_OF_VARIABLE }` 中的变量。`%{ NAME_OF_VARIABLE }` 中的 *NAME_OF_VARIABLE* 是一种取自下面列表的字符串：
- **HTTP**报头：

```
HTTP\_USER\_AGENT
HTTP\_REFERER
HTTP\_COOKIE
HTTP\_FORWARDED
HTTP\_HOST
HTTP\_PROXY\_CONNECTION
HTTP\_ACCEPT
```

- 连接与请求：

```
REMOTE\_ADDR
REMOTE\_HOST
REMOTE\_PORT
REMOTE\_USER
REMOTE\_IDENT
REQUEST\_METHOD
SCRIPT\_FILENAME
REQUEST\_PATH
CONTEXT\_PATH
SERVLET\_PATH
PATH\_INFO
QUERY\_STRING
AUTH\_TYPE
```

- 服务器内部：

```
DOCUMENT\_ROOT
SERVER\_NAME
SERVER\_ADDR
SERVER\_PORT
```

```
SERVER\_PROTOCOL
SERVER\_SOFTWARE
```

- 日期与时间：

```
TIME\_YEAR
TIME\_MON
TIME\_DAY
TIME\_HOUR
TIME\_MIN
TIME\_SEC
TIME\_WDAY
TIME
```

- 特殊字符串：

```
THE\_REQUEST
REQUEST\_URI
REQUEST\_FILENAME
HTTPS
```

这些变量对应着相似名称的 HTTP MIME 报头和 Servlet API 方法。多数都记录在各种手册和 CGI 规范中。下面列出了重写 Valve 专有的那些变量：

- `REQUEST_PATH`
  对应用于映射的完整路径。
- `CONTEXT_PATH`
  对应映射的上下文的路径。
- `SERVLET_PATH`
  对应 Servlet 路径。
- `THE_REQUEST`
  由浏览器发送给服务器的完整 HTTP 请求代码行（比如，`GET /index.html HTTP/1.1`）。这并不包括任何由浏览器发送的额外报头。
- `REQUEST_URI`
  HTTP 请求代码行中所请求的资源（在上例中，应为为 `/index.html`）。
- `REQUEST_FILENAME`
  与请求相匹配的文件或脚本的完整本地文件系统路径。
- `HTTPS`
  当连接使用 SSL/TLS 时，含有文本 `"on"`，否则含有 `"off"`。

另外还需要注意的是：

1. `SCRIPT_FILENAME` 和 `REQUEST_FILENAME` 含有同样的值：Apache 服务器内部结构 `request_rec` 的 `filename` 字段值。第一个名称常被称为 CGI 变量值，第二个名称相当于 `REQUEST_URI`（包含 `request_rec` 的 `uri` 字段值）。
2. `%{ENV:variable}`。其中的 `variable` 可以是任何 Java 系统属性。目前可以使用。
3. `%{SSL:variable}`。其中的 `variable` 是 SSL 环境变量名。目前还未实现。范例：`%{SSL:SSL_CIPHER_USEKEYSIZE}` 可能扩展到 `128`。
4. `%{HTTP:header}`。其中的 `header` 可以是任意的 HTTP MIME 报头名称。该变量常用来获取发送到 HTTP 请求的报头值。范例：`%{HTTP:Proxy-Connection}` 是 HTTP 报头 `Proxy-Connection:` 的值。

CondPattern 即条件模式，是一种应用于 *TestString* 当前实例的正则表达式。*TestString* 在匹配 *CondPattern* 之前，会首先求值。

谨记：*CondPattern* 是一种兼容 perl 并带有一些扩展的正则表达式。

1. 可以在模式字符串前加上 `!` 字符作为前缀，来指定非匹配模式。

2. *CondPattern* 有一些特殊变体。除了真正的正则表达式字符串外，还可以使用下列组合形式之一：

- `<CondPattern` (字母顺序高于)
将 *CondPattern* 当成一种纯粹字符串，按照字母顺序将其与 *TestString* 进行对比。如果在字母顺序上，*TestString* 高于 *CondPattern* ，则为 true。
- `>CondPattern` （字母顺序低于） 将 *CondPattern* 当成一种纯粹字符串，按照字母顺序将其与 *TestString* 进行对比。如果在字母顺序上，*TestString* 低于 *CondPattern* ，则为 true。
- `=CondPattern'` (字母顺序等于)
将 *CondPattern* 当成一种纯粹字符串，按照字母顺序将其与 *TestString* 进行对比。如果在字母顺序上，*TestString* 等于 *CondPattern* ，则为 true。 - `-d` （目录）
将 `TestString` 当成一种路径名，测试它是否存在并且是一个目录。
- `-f` （常规文件） 将 `TestString` 当成一种路径名，测试它是否存在并且是一个常规文件。
- `-s` （常规文件，带有文件尺寸）
将 `TestString` 当成一种路径名，测试它是否存在并且是一个文件大小大于 0 的普通文件。

注意：所有这些测试都可以加上前缀 `!` 来使它们的含义反向。 3. 可以将 `[flag]` 做为 `RewriteCond` 指令的第三个参数，为 `CondPattern` 设定标记。这里的 flag 是一个包含下列任一标记，且标记间由逗号分隔的列表：
- `nocase|NC` （不区分大小写） 无论是扩展的 *TestString* 还是在 *CondPattern* 中，都不区分大小写（A-Z 和 a-z 是等同的）。该标记只有在比较 *TestString* 和 *CondPattern* 时才是有效的。对于文件系统和子请求（HTTP 请求）是无效的。 - `ornext|OR` （或者下一个条件） 利用本地 OR（而不是隐式的 AND）来组合规则条件。典型范例为：

```
RewriteCond %{REMOTE_HOST}  ^host1.*  [OR]
RewriteCond %{REMOTE_HOST}  ^host2.*  [OR]
RewriteCond %{REMOTE_HOST}  ^host3.*
RewriteRule ...some special stuff for any of these hosts...
```

没有该标记，你必须写三遍条件/规则对。

范例：

假如想根据请求头的 `User-Agent:` 对网站主页进行重写，可以使用下列代码：

```
RewriteCond  %{HTTP_USER_AGENT}  ^Mozilla.*
RewriteRule  ^/$                 /homepage.max.html  [L]

RewriteCond  %{HTTP_USER_AGENT}  ^Lynx.*
RewriteRule  ^/$                 /homepage.min.html  [L]

RewriteRule  ^/$                 /homepage.std.html  [L]
```

说明：如果使用的浏览器将它自身标识为 'Mozilla'（包括 Netscape Navigator、Mozilla，等等），那么这就是内容最大化主页（max homepage。它可以包含框架或其他特性）；如果使用的是 Lynx 浏览器（基于终端的），那么获得的就是内容最小化的主页（min homepage）——专为便于浏览文本而设计的版本；如果这些条件都不适用（你使用的是其他浏览器，或者你的浏览器将其自身标识为非标准内容），那么得到的就是标准主页（std homepage）。

2. RewriteMap

格式：`RewriteMap name rewriteMapClassName optionalParameters`。

通过用户必须实现的一个接口来实现映射。类名为 `org.apache.catalina.valves.rewrite.RewriteMap`，代码为：

```
package org.apache.catalina.valves.rewrite;

public interface RewriteMap {
public String setParameters(String params);
```

```
public String lookup(String key);
}
```

3. RewriteRule

格式：`RewriteRule Pattern Substitution`

`RewriteRule` 指令是重写机制的核心。此指令可以多次使用，每个实例都定义一个单独的重写规则。这些规则的定义顺序尤为重要，因为这是在运行时应用它们的顺序。

模式是一个作用于当前 URL 的兼容 perl 的正则表达式，这里的“当前”是指该规则生效时的 URL，它可能与被请求的 URL 不同，因为其他规则可能在此之前已经发生匹配并对它做了改动。

下面是关于正则表达式格式的一些提示：

> 文本：
>
> - `.`——匹配任何单个字符
> - `[chars]`——匹配当前字符
> - `[^chars]`——不匹配当前字符
> - `text1|text2`——匹配 text1 或 text2

> 量词：- `?`——零个或者 1 个 `?` 号前的字符 - `*`——零个或者 N 个 `*` 号前的字符（N > 0） - `+`——零个或 N 个 `+` 号前的字符（N > 1）

> 分组：`(text)`——文本分组（设定第 N 组可以被引用为 RewriteRule）

> 行锚
>
> `^`——匹配一行起始处的空字符串。
> `$`——匹配一行结束处的空字符串。

> 转义
>
> `\char`——将指定字符转义（比如将`.`、`[]`、`()` 等字符转义）

要想了解更多关于正则表达式的信息，请参见 perl 正则表达式在线联机手册（perldoc perlre）。关于正则表达式及其变体（POSIX 正则表达式）的详情，可看看这本书：

```
*Mastering Regular Expressions, 2nd Edition* （目前该书为第 3 版）
Jeffrey E.F. Friedl
O'Reilly & Associates, Inc. 2002
ISBN 978-0-596-00289-3
```

在规则中，NOT 字符（`!`）可作为模式的前缀，实现逆向模式。比如“如果当前 URL 并不匹配该模式”。这可以用在易于匹配的是逆向模式这种异常情况下，或者也可以作为最后一个默认规则来使用。

注意：在使用 NOT 字符反转模式时，不能在模式中包含分组的通配成分。这是因为，如果模式不匹配（比如反匹配），分组内将没有内容。如果使用了逆向模式，就不能在替代字符串中使用 `$N`。

重写规则中的替代字符串（substitution string）是一种用来取代模式匹配的原始 URL 的字符串。除了纯文本之外，它还包括：

1. 反向引用（`$N`）RewriteRule 模式。
2. 反向引用（`%N`）最后匹配的 RewriteCond 模式。

3. 规则条件测试字符串中（`%{VARNAME}`）的服务器变量。
4. 映射函数调用（`${mapname:key|default}`）。

反向引用表示的是形式 `$N`（N 的范围为 0-9），它是指用模式所匹配的第 **N** 组的内容去替换 URL。服务器变量 `RewriteCond` 指令的 *TestString* 所用的相同。映射函数来自 `RewriteMap` 指令。这 3 类变量都按照上述顺序展开。

如前所述，所有的重写规则都应用于替代字符串（按照配置文件中所定义的顺序）。URL 完全由替代字符串所替换，直到所有规则都应用完毕，重写过程才结束（或利用 `L` 标记来终止）。

还有一个特殊的替代字符串：`-`，意思是不替代，当需要让重写规则只匹配 URL 而不替换时，就用得上它了。这一字符串通常与 **C**（chain）标记配合使用，为的是在替换发生前应用多个模式。

另外，还可以将 `[`flags`]` 做为 RewriteRule 的第三个参数，从而为替代字符串设置特殊标记。*flags*是一个包含下列标记且标记间以逗号分隔的列表：

- `chain|C`（与下一个规则相链接）

此标记使当前规则与下一个规则（它又可以与其后规则相链接，如此反复）相链接。 它产生如下效果：如果某个规则被匹配，通常会继续处理其后继规则，这个标记就不起作用；如果某规则不被匹配，则其后继链接规将会被忽略。比如，在执行一个外部重定向时， 对一个目录级规则集，你可能需要删除 `.www`（因为此处不应该出现 `.www`）。

- `cookie|CO` = ***NAME:VAL:domain[:lifetime[:path]]*** （设置 cookie）

在客户端浏览器上设置一个 cookie。cookie 的名称是 ***NAME***，其值是 ***VAL***。 *domain* 字段是该 cookie 的域，比如 `.apache.org`，可选的*lifetime* 是 cookie 生命周期（以分钟计），可选的 *path* 是 cookie 的路径。

- `env|E` = *VAR:VAL* （设置环境变量）

强制一个名为 *VAR* 的请求变量值为 *VAL*，VAL可以包含可扩展的反向引用的正则表达式 `$N` 和 `%N`。可以多次使用该标记，以便设置多个变量。

- `forbidden|F`（强制禁止访问该 URL ）

强制禁止访问当前的 URL——立即反馈一个 HTTP 响应代码 403。使用这个标记，可以链接若干 `RewriteConds` 以阻断某些 URL。

- `gone|G`（强制 URL 为已失效）

强制当前 URL 为已失效——立即反馈一个 HTTP 响应代码 410（请求资源已被删除）。使用这个标记，可以标明页面已被删除而不存在.

- `host|H=Host`（重写虚拟主机）
  重写虚拟主机，而不是重写 URL。

- `last|L`（最后一个规则）

立即停止重写操作，并不再应用其他重写规则。它对应于 Perl 中的 `last` 命令或 C 语言中的 `break` 命令。使用该标记可以防止当前已被重写的 URL 被后续规则所继续重写。比如，用它可以将根路径的 URL（`/`）重写为实际存在的 URL，比如：`/e/www/`。

- `next|N`（重新执行）

重新执行重写操作（从第一个重写规则重新开始）。这时，要匹配的 URL 已不是原始 URL 了，而是经最后一个重写规则处理过的 URL。它对应于 Perl 中的 `next` 命令或 C 语言中的 `continue` 命令。此标记可以重新开始重写操作，立即回到循环的头部。

但是要小心，不要制造死循环！

- `nocase|NC`（不区分大小写）

使模式不区分大小写。当模式与当前 URL 匹配时，A-Z 和 a-z 没有区别。

- `noescape|NE`（在输出中不对 URI 进行转义）

此标记阻止重写 Valve 对重写结果应用常规的 URI 转义规则。一般情况下，特殊字符（如`%`、`$`、`;`等）都会被转义为十六进制值。此标记可以阻止这种转义，允许百分号等符号出现在输出中，如：

```
RewriteRule /foo/(.*) /bar?arg=P1\%3d$1 [R,NE]
```

将 `/foo/zed` 转化为 `/bar?arg=P1=zed` 的安全请求。

- `qsappend|QSA`（附加查询串）

该标记会强制重写引擎将替代字符串中的一个查询字符串部分添加到已有字符串上，而不是简单地替换已有字符串。当你想要通过重写规则为查询字符串添加更多数据时，可以使用该标记。

- `redirect|R`**[=code]**（强制重定向）

将 `http://thishost[:thisport]/`（使新的 URL 成为一个 URI）作为替代字符串的前缀，从而强制执行一个外部重定向。如果没有指定 code，则产生一个HTTP 响应代码 302（暂时移动）。如果需要使用在 300-400 范围内的其他响应代码，只需在此指定这个数值即可。另外，还可以使用下列符号名称：`temp`（默认），`permanent`、`seeother`。用它可以把规范化的 URL 反馈给客户端，如重写 `/~` 为 `/u/`，或对 `/u/user` 加上斜杠，等等。

注意：在使用这个标记时，必须确保替换字段是一个有效的 URL！否则它会指向一个无效的位置！并且要记住，此标记本身只是对 URL 加上 `http://thishost[:thisport]/` 前缀而已，不会妨碍重写操作。通常，你会希望停止重写，然后立即重定向。要想停止重写，你还需要添加 `L` 标记。

- `skip|S`**=num**（忽略后续规则）

如果当前规则匹配，此标记会强制重写引擎跳过当前匹配规则后面的 num 个规则。它可以实现一个类似 if-then-else 的构造：then 子句的最后一个规则是 `skip = N`，其中 N 代表 else 子句中的规则数目。（该标记不同于 `chain|C` 标记！）

- `type|T`**=MIME-type**（强制指定 MIME 类型）

强制指定目标文件的 MIME 类型为 **MIME-type**，可基于一些条件设置内容类型。比如在下面的代码段中，如果利用 `.phps` 扩展调用 `.php` 文件，它们就能被 `mod_php` 模块显示。

```
RewriteRule ^(.+\.php)s$ $1 [T=application/x-httpd-php-source]
```

目录

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com