Eclipse OSGi

0 尤埃与产品简介

西安尤埃信息技术有限公司（http://www.uishell.com）成立于 2008 年 5 月份，专注于尤埃开放服务平台和尤埃 SaaS 引擎云计算产品开发。

尤埃开放服务平台（XAUI Open Service Platform，UIOSP）是一个移植了 **OSGi** 规范的**动态插件化与模块化平台**，支持插件化与模块化、SOA 和模块扩展。

尤埃 SaaS 引擎（XAUI SaaS Engine，XSE）是一个 SaaS 应用商店**开放平台**。该平台是面向 SaaS 运营商、SaaS 提供商和 SaaS 消费者三个角色的 **PaaS 云计算平台**，其模式为“SaaS 运营商负责平台运营，SaaS 提供商利用平台提供的开发工具包基于 VS2008SP1 开发 SaaS 应用并上传，SaaS 消费者在应用商店挑选、购买并使用 SaaS 应用”。该平台由应用商店网站、应用开发工具包和应用虚拟运行环境构成。

1 osgi.framework

1.1 Bundle 相关

1.1.1 Bundle

Bundle 接口表示在框架中安装的一个 Bundle。每一个 Bundle 有一个由框架设置的唯一标识。它提供以下功能：

（1）Bundle 操作，启动、停止、更新、卸载操作及 Bundle 状态。

（2）Bundle 信息，头信息、ID、Location、特征名称和上次更新时间。

（3）类和资源加载，获取服务引用。

（4）Bundle 上下文。

```
package org.osgi.framework;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.Dictionary;
import java.util.Enumeration;

public interface Bundle {
    //已卸载状态。
    public static final int  UNINSTALLED          = 0x00000001;
    //已安装状态。
    public static final int  INSTALLED            = 0x00000002;
    //已解析状态。
    public static final int  RESOLVED             = 0x00000004;
    //正在启动状态。
    public static final int  STARTING             = 0x00000008;
    //正在停止状态。
    public static final int  STOPPING             = 0x00000010;
    //激活状态。
    public static final int  ACTIVE               = 0x00000020;
    //Bundle的autostart设置影响启动策略，这个持久设置可有：
    //（1）Stopped——Bundle不可以被启动；
    //（2）Started with eager activation——启动后立即激活；
    //（3）Started with declared activation——启动后，第一个类加载时激活。
    //这个值用于start函数，表示启动后立即激活，但不更改autostart设置。
    public static final int  START_TRANSIENT          = 0x00000001;
    //这个值用于start函数，表示使用Bundle-ActivationPolicy来启动，并设置
    //autostart为Started with declared activation。
    public static final int  START_ACTIVATION_POLICY  = 0x00000002;
    //用于stop函数，表示停止后不更改autostart设置。
    public static final int  STOP_TRANSIENT           = 0x00000001;
```

```
//返回Bundle当前状态。
public int getState();
//使用参数启动Bundle。这个参数可以是：
//(1)0——启动后立即激活，并设置autostart为Started with eager activation。
//（2）START_TRANSIENT——同0，但不更改autostart。
//（3）START_ACTIVATION_POLICY——启动后根据Bundle-ActivationPolicy来
//激活Bundle，并更改autostart为Started with declared activation。
//（4）START_ACTIVATION_POLICY | START_TRANSIENT——同（3），但不更改
//autostart设置。
//如果Bundle状态时UNINSTALLED，则抛出IllegalStateException；
//如果框架实现了可选的启动级别且当前启动级别小于Bundle启动级别，
//若设置START_TRANSIENT，则抛出BundleException。
public void start(int options) throws BundleException;
//相当于start(0)。
public void start() throws BundleException;
//停止一个Bundle，参数options可为：
//（1）0——停止并设置autostart为stopped。
//（2）STOP_TRANSIENT——停止，不更改autostart。
public void stop(int options) throws BundleException;
//相当于stop(0)。
public void stop() throws BundleException;
//更新Bundle。
public void update() throws BundleException;
public void update(InputStream in) throws BundleException;
//卸载一个Bundle。
public void uninstall() throws BundleException;
//获取这个Bundle的头信息。
public Dictionary getHeaders();
//Bundle ID。
public long getBundleId();
```

```
    //安装位置。
    public String getLocation();
    //获取这个Bundle注册的服务。
    public ServiceReference[] getRegisteredServices();
    //获取这个Bundle使用的服务。
    public ServiceReference[] getServicesInUse();
    public boolean hasPermission(Object permission);
    //获取资源。
    public URL getResource(String name);
    //获取本地化的头信息。
    public Dictionary getHeaders(String locale);
    //特征名。
    public String getSymbolicName();
    //装载类。
    public Class loadClass(String name) throws ClassNotFoundException;
    //获取所有资源。
    public Enumeration getResources(String name) throws IOException;
    public Enumeration getEntryPaths(String path);
    public URL getEntry(String path);
    //上次更新时间。
    public long getLastModified();
    //查找资源。
    public Enumeration findEntries(String path, String filePattern,
            boolean recurse);
    //获取上下文。
    public BundleContext getBundleContext();
}
```

1.1.2 BundleActivator

自定义启动和停止 Bundle。当 Bundle 在 Resolved<->Active 之间转换时，调用 start 和 stop。

```
package org.osgi.framework;


public interface BundleActivator {
    public void start(BundleContext context) throws Exception;
    public void stop(BundleContext context) throws Exception;
}
```

1.1.3 BundleContext

BundleContext 用于使关联 Bundle 与系统的其它功能进行交互。它提供的功能包括：

（1）获取属性。

（2）检索和安装 Bundle。

（3）注册和获取服务引用。

（4）服务、Bundle 和框架的事件与监听器。

（5）过滤器和持久存储。

```
package org.osgi.framework;


import java.io.File;
import java.io.InputStream;
import java.util.Dictionary;


public interface BundleContext {
    //系统属性。
    public String getProperty(String key);
    //返回关联的Bundle。
```

```
public Bundle getBundle();
//安装一个Bundle。
public Bundle installBundle(String location)
        throws BundleException;
public Bundle installBundle(String location, InputStream input)
        throws BundleException;
//获取指定ID的Bundle。
public Bundle getBundle(long id);
//获取所有安装的Bundle。
public Bundle[] getBundles();
//服务监听器。
public void addServiceListener(ServiceListener listener,
        String filter) throws InvalidSyntaxException;
public void addServiceListener(ServiceListener listener);
public void removeServiceListener(ServiceListener listener);
//Bundle监听器。
public void addBundleListener(BundleListener listener);
public void removeBundleListener(BundleListener listener);
//Framework监听器。
public void addFrameworkListener(FrameworkListener listener);
public void removeFrameworkListener(FrameworkListener listener);
//注册服务。
public ServiceRegistration registerService(String[] clazzes,
        Object service, Dictionary properties);
public ServiceRegistration registerService(String clazz,
        Object service, Dictionary properties);
//获取服务。
public ServiceReference[] getServiceReferences(String clazz,
        String filter) throws InvalidSyntaxException;
public ServiceReference[] getAllServiceReferences(String clazz,
```

```
            String filter) throws InvalidSyntaxException;
    public ServiceReference getServiceReference(String clazz);
    public Object getService(ServiceReference reference);
    //取消使用服务。
    public boolean ungetService(ServiceReference reference);
    //在持久化存储中创建一个文件。如果不支持文件系统，则返回null。
    public File getDataFile(String filename);
    //创建过滤器。
    public Filter createFilter(String filter)
            throws InvalidSyntaxException;
}
```

1.1.4 BundleEvent

描述 Bundle 声明周期变更的事件。需要注意的是，这个事件是异步传输的。

```
package org.osgi.framework;


import java.util.EventObject;


public class BundleEvent extends EventObject {
    //变更的Bundle。
    private final Bundle bundle;
    //声明周期变更类型。
    private final int    type;
    //类型：已安装、已启动、已停止、已更新、已卸载、已解析、未解析、
    //正在启动、正在停止、将被晚激活。
    public final static int  INSTALLED         = 0x00000001;
    public final static int  STARTED           = 0x00000002;
    public final static int  STOPPED           = 0x00000004;
    public final static int  UPDATED           = 0x00000008;
```

```
    public final static int UNINSTALLED       = 0x00000010;
    public final static int RESOLVED          = 0x00000020;
    public final static int UNRESOLVED        = 0x00000040;
    public final static int STARTING          = 0x00000080;
    public final static int STOPPING          = 0x00000100;
    public final static int LAZY_ACTIVATION   = 0x00000200;

    public BundleEvent(int type, Bundle bundle) {
        super(bundle);
        this.bundle = bundle;
        this.type = type;
    }

    public Bundle getBundle() {
        return bundle;
    }

    public int getType() {
        return type;
    }
}
```

1.1.5 BundleException

一个 BundleException 用于指示发生了一个 Bundle 生命周期问题。

```
package org.osgi.framework;

public class BundleException extends Exception {
    static final long serialVersionUID = 3571095144220455665L;
```

```
    private final Throwable  cause;

    public BundleException(String msg, Throwable cause) {
        super(msg);
        this.cause = cause;
    }
    public BundleException(String msg) {
        super(msg);
        this.cause = null;
    }
    public Throwable getNestedException() {
        return cause;
    }
    public Throwable getCause() {
        return cause;
    }
    public Throwable initCause(Throwable cause) {
        throw new IllegalStateException();
    }
}
```

1.1.6 BundleListener

```
package org.osgi.framework;

import java.util.EventListener;

public interface BundleListener extends EventListener {
    public void bundleChanged(BundleEvent event);
}
```

1.1.7 SynchronousBundleListener

同步处理，即 Bundle 的生命周期处理中必须同步的处理完监听器后，才可以执行其它操作。

```
package org.osgi.framework;


public interface SynchronousBundleListener extends BundleListener {
}
```

1.2 服务相关

1.2.1 ServiceReference

表示对一个服务的引用。BundleContext.getServiceReference 将返回该对象。它可以在 Bundle 间共享，用于检索服务属性和获取服务对象。在框架注册的每一个服务有一个唯一的 ServiceRegistration 和多个引用它的 ServiceReference 对象。关联相同 ServiceRegistration 的服务引用有相同的哈希值。

```
package org.osgi.framework;


public interface ServiceReference extends Comparable {
    //获取服务的属性。
    public Object getProperty(String key);
    //获取所有属性键。
    public String[] getPropertyKeys();
    //获取注册服务的Bundle。
    public Bundle getBundle();
    //获取通过这个服务引用使用服务的Bundle。
    public Bundle[] getUsingBundles();
    //判断注册当前服务引用对应的服务的Bundle和指定的Bundle，
    //对于指定的类是否使用相同的包。
```

```
    public boolean isAssignableTo(Bundle bundle, String className);
    //比较。
    public int compareTo(Object reference);
}
```

1.2.2 ServiceRegistration

表示一个已经被注册的服务。当 BundleContext.registerService 方法调用成功时，框架返回一个该对象。这个对象由注册服务的 Bundle 拥有且不与其它 Bundle 共享。它用于更新服务和卸载服务。

```
package org.osgi.framework;

import java.util.Dictionary;

public interface ServiceRegistration {
    //获取一个引用。
    public ServiceReference getReference();
    //设置服务属性。
    public void setProperties(Dictionary properties);
    //卸载服务。
    public void unregister();
}
```

1.2.3 ServiceFactory

允许在 OSGi 环境提供自定义的服务对象。当注册服务时，一个 ServiceFactory 可以用于替代服务对象。可以根据需要返回真正的服务对象，类似于服务代理。

```
package org.osgi.framework;

public interface ServiceFactory {
    //获取服务对象，将被缓存。
```

```
    public Object getService(Bundle bundle,
        ServiceRegistration registration);
    //释放一个服务对象。
    public void ungetService(Bundle bundle,
        ServiceRegistration registration,  Object service);
}
```

1.2.4 ServiceEvent

用于描述服务生命周期的变更。这个事件被同步传输。

```
package org.osgi.framework;

import java.util.EventObject;

public class ServiceEvent extends EventObject {
    private final ServiceReference  reference;
    private final int               type;
    public final static int         REGISTERED        = 0x00000001;
    public final static int         MODIFIED          = 0x00000002;
    public final static int         UNREGISTERING     = 0x00000004;

    public ServiceEvent(int type, ServiceReference reference) {
        super(reference);
        this.reference = reference;
        this.type = type;
    }
    public ServiceReference getServiceReference() {
        return reference;
    }
    public int getType() {
```

```
        return type;
    }
}
```

1.2.5 ServiceListener

监听器会根据权限过滤。

```
package org.osgi.framework;

import java.util.EventListener;

public interface ServiceListener extends EventListener {
    public void serviceChanged(ServiceEvent event);
}
```

1.2.6 AllServicesListener

```
package org.osgi.framework;

public interface AllServiceListener extends ServiceListener {

}
```

1.3 Framework 相关

1.3.1 FrameworkUtil

```
package org.osgi.framework;

import org.eclipse.osgi.framework.internal.core.FilterImpl;
```

```
public class FrameworkUtil {
    private FrameworkUtil() {}
    //创建一个过滤器。
    public static Filter createFilter(String filter)
            throws InvalidSyntaxException
    {
        return new FilterImpl(filter);
    }
}
```

1.3.2 FrameworkEvent

框架的普通事件，它是一个异步事件。

```
package org.osgi.framework;

import java.util.EventObject;

public class FrameworkEvent extends EventObject {
    private final Bundle bundle;
    private final Throwable  throwable;
    private final int    type;
    public final static int  STARTED              = 0x00000001;
    public final static int  ERROR             = 0x00000002;
    public final static int  PACKAGES_REFRESHED   = 0x00000004;
    public final static int  STARTLEVEL_CHANGED   = 0x00000008;
    public final static int  WARNING              = 0x00000010;
    public final static int  INFO             = 0x00000020;

    public FrameworkEvent(int type, Object source)
    {
```

```
        super(source);
        this.type = type;
        this.bundle = null;
        this.throwable = null;
    }
    public FrameworkEvent(int type, Bundle bundle, Throwable throwable)
    {
        super(bundle);
        this.type = type;
        this.bundle = bundle;
        this.throwable = throwable;
    }
    public Throwable getThrowable() {
        return throwable;
    }
    public Bundle getBundle() {
        return bundle;
    }
    public int getType() {
        return type;
    }
}
```

1.3.3 FrameworkListener

```
package org.osgi.framework;

import java.util.EventListener;

public interface FrameworkListener extends EventListener {
```

```
    public void frameworkEvent(FrameworkEvent event);
}
```

1.4 其它

1.4.1 AdminPermission（略）

1.4.2 PackagePermission（略）

1.4.3 ServicePermission（略）

1.4.4 Constants

```
package org.osgi.framework;


public interface Constants {
    public static final String  SYSTEM_BUNDLE_LOCATION
        = "System Bundle";
    public static final String  SYSTEM_BUNDLE_SYMBOLICNAME
        = "system.bundle";
    public static final String  BUNDLE_CATEGORY
        = "Bundle-Category";
    public static final String  BUNDLE_CLASSPATH
        = "Bundle-ClassPath";
    public static final String  BUNDLE_COPYRIGHT
        = "Bundle-Copyright";
    public static final String  BUNDLE_DESCRIPTION
        = "Bundle-Description";
    public static final String  BUNDLE_NAME
        = "Bundle-Name";
    public static final String  BUNDLE_NATIVECODE
```

```
    = "Bundle-NativeCode";
public static final String  EXPORT_PACKAGE
    = "Export-Package";
public static final String  EXPORT_SERVICE
    = "Export-Service";
public static final String  IMPORT_PACKAGE
    = "Import-Package";
public static final String  DYNAMICIMPORT_PACKAGE
    = "DynamicImport-Package";
public static final String  IMPORT_SERVICE
    = "Import-Service";
public static final String  BUNDLE_VENDOR
    = "Bundle-Vendor";
public static final String  BUNDLE_VERSION
    = "Bundle-Version";
public static final String  BUNDLE_DOCURL
    = "Bundle-DocURL";
public static final String  BUNDLE_CONTACTADDRESS
    = "Bundle-ContactAddress";
public static final String  BUNDLE_ACTIVATOR
    = "Bundle-Activator";
public static final String  BUNDLE_UPDATELOCATION
    = "Bundle-UpdateLocation";


public static final String  PACKAGE_SPECIFICATION_VERSION
    = "specification-version";
public static final String  BUNDLE_NATIVECODE_PROCESSOR
    = "processor";
public static final String  BUNDLE_NATIVECODE_OSNAME
    = "osname";
```

```
public static final String  BUNDLE_NATIVECODE_OSVERSION
    = "osversion";
public static final String  BUNDLE_NATIVECODE_LANGUAGE
    = "language";
public static final String  BUNDLE_REQUIREDEXECUTIONENVIRONMENT
    = "Bundle-RequiredExecutionEnvironment";
public static final String  FRAMEWORK_VERSION
    = "org.osgi.framework.version";
public static final String  FRAMEWORK_VENDOR
    = "org.osgi.framework.vendor";
public static final String  FRAMEWORK_LANGUAGE
    = "org.osgi.framework.language";
public static final String  FRAMEWORK_OS_NAME
    = "org.osgi.framework.os.name";
public static final String  FRAMEWORK_OS_VERSION
    = "org.osgi.framework.os.version";
public static final String  FRAMEWORK_PROCESSOR
    = "org.osgi.framework.processor";
public static final String  FRAMEWORK_EXECUTIONENVIRONMENT
    = "org.osgi.framework.executionenvironment";
public static final String  FRAMEWORK_BOOTDELEGATION
    = "org.osgi.framework.bootdelegation";
public static final String  FRAMEWORK_SYSTEMPACKAGES
    = "org.osgi.framework.system.packages";
public static final String  SUPPORTS_FRAMEWORK_EXTENSION
    = "org.osgi.supports.framework.extension";
public static final String  SUPPORTS_BOOTCLASSPATH_EXTENSION
    = "org.osgi.supports.bootclasspath.extension";
public static final String  SUPPORTS_FRAMEWORK_FRAGMENT
    = "org.osgi.supports.framework.fragment";
```

```
public static final String  SUPPORTS_FRAMEWORK_REQUIREBUNDLE
    = "org.osgi.supports.framework.requirebundle";
public static final String  OBJECTCLASS
    = "objectClass";
public static final String  SERVICE_ID
    = "service.id";
public static final String  SERVICE_PID
    = "service.pid";
public static final String  SERVICE_RANKING
    = "service.ranking";
public static final String  SERVICE_VENDOR
    = "service.vendor";
public static final String  SERVICE_DESCRIPTION
    = "service.description";
public final static String  BUNDLE_SYMBOLICNAME
    = "Bundle-SymbolicName";
public final static String  SINGLETON_DIRECTIVE
    = "singleton";
public final static String  FRAGMENT_ATTACHMENT_DIRECTIVE
    = "fragment-attachment";
public final static String  FRAGMENT_ATTACHMENT_ALWAYS
    = "always";
public final static String  FRAGMENT_ATTACHMENT_RESOLVETIME
    = "resolve-time";
public final static String  FRAGMENT_ATTACHMENT_NEVER
    = "never";
public final static String  BUNDLE_LOCALIZATION
    = "Bundle-Localization";
public final static String  BUNDLE_LOCALIZATION_DEFAULT_BASENAME
    = "OSGI-INF/l10n/bundle";
```

```
public final static String  REQUIRE_BUNDLE
    = "Require-Bundle";
public static final String  BUNDLE_VERSION_ATTRIBUTE
    = "bundle-version";
public final static String  FRAGMENT_HOST
    = "Fragment-Host";
public final static String  SELECTION_FILTER_ATTRIBUTE
    = "selection-filter";
public final static String  BUNDLE_MANIFESTVERSION
    = "Bundle-ManifestVersion";
public final static String  VERSION_ATTRIBUTE
    = "version";
public final static String  BUNDLE_SYMBOLICNAME_ATTRIBUTE
    = "bundle-symbolic-name";
public final static String  RESOLUTION_DIRECTIVE
    = "resolution";
public final static String  RESOLUTION_MANDATORY
    = "mandatory";
public final static String  RESOLUTION_OPTIONAL
    = "optional";
public final static String  USES_DIRECTIVE
    = "uses";
public final static String  INCLUDE_DIRECTIVE
    = "include";
public final static String  EXCLUDE_DIRECTIVE
    = "exclude";
public final static String  MANDATORY_DIRECTIVE
    = "mandatory";
public final static String  VISIBILITY_PRIVATE
    = "private";
```

```
    public final static String  VISIBILITY_REEXPORT
        = "reexport";
    public final static String  EXTENSION_DIRECTIVE
        = "extension";
    public final static String  EXTENSION_FRAMEWORK
        = "framework";
    public final static String  EXTENSION_BOOTCLASSPATH
        = "bootclasspath";
    public final static String  BUNDLE_ACTIVATIONPOLICY
        = "Bundle-ActivationPolicy";
    public final static String  ACTIVATION_LAZY
        = "lazy";

}
```

1.4.5 Filter

基于 RFC1960 的过滤器。

```
package org.osgi.framework;

import java.util.Dictionary;
public interface Filter {
    public boolean match(ServiceReference reference);
    public boolean match(Dictionary dictionary);
    public String toString();
    public boolean equals(Object obj);
    public int hashCode();
    public boolean matchCase(Dictionary dictionary);
}
```

1.4.6 InvalidSyntaxException

```
package org.osgi.framework;


public class InvalidSyntaxException extends Exception {
    private final String filter;
    private final Throwable  cause;


    public InvalidSyntaxException(String msg, String filter) {
        super(msg);
        this.filter = filter;
        this.cause = null;
    }
    public InvalidSyntaxException(String msg, String filter,
        Throwable cause) {
        super(msg);
        this.filter = filter;
        this.cause = cause;
    }
    public String getFilter() {
        return filter;
    }
    public Throwable getCause() {
        return cause;
    }
    public Throwable initCause(Throwable cause) {
        throw new IllegalStateException();
    }
}
```

1.4.7 Version

表示 Bundle 和包的版本。由 Major、Minor、Micro、Qualifier 组成。

```
package org.osgi.framework;


import java.util.NoSuchElementException;
import java.util.StringTokenizer;


public class Version implements Comparable {
    private final int        major;
    private final int        minor;
    private final int        micro;
    private final String     qualifier;
    private static final String SEPARATOR     = ".";


    public static final Version emptyVersion  = new Version(0, 0, 0);


    public Version(int major, int minor, int micro) {
        this(major, minor, micro, null);
    }


    public Version(int major, int minor, int micro, String qualifier) {
        if (qualifier == null) {
            qualifier = "";
        }


        this.major = major;
        this.minor = minor;
        this.micro = micro;
        this.qualifier = qualifier;
```

```
        validate();
    }
    //解析字符串。
    public Version(String version) {
        int major = 0;
        int minor = 0;
        int micro = 0;
        String qualifier = "";

        try {
            StringTokenizer st = new StringTokenizer(
                version, SEPARATOR, true);
            major = Integer.parseInt(st.nextToken());

            if (st.hasMoreTokens()) {
                st.nextToken(); // consume delimiter
                minor = Integer.parseInt(st.nextToken());

                if (st.hasMoreTokens()) {
                    st.nextToken(); // consume delimiter
                    micro = Integer.parseInt(st.nextToken());

                    if (st.hasMoreTokens()) {
                        st.nextToken(); // consume delimiter
                        qualifier = st.nextToken();

                        if (st.hasMoreTokens()) {
                            throw new IllegalArgumentException(
                                "invalid format");
                        }
```

```
                }
            }
        }
    }
    catch (NoSuchElementException e) {
        throw new IllegalArgumentException("invalid format");
    }

    this.major = major;
    this.minor = minor;
    this.micro = micro;
    this.qualifier = qualifier;
    validate();
}
//验证有效性。
private void validate() {
    if (major < 0) {
        throw new IllegalArgumentException("negative major");
    }
    if (minor < 0) {
        throw new IllegalArgumentException("negative minor");
    }
    if (micro < 0) {
        throw new IllegalArgumentException("negative micro");
    }
    int length = qualifier.length();
    for (int i = 0; i < length; i++) {
        if ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcde
            fghijklmnopqrstuvwxyz0123456789_-".
            indexOf(qualifier.charAt(i)) == -1) {
```

```
                throw new IllegalArgumentException("invalid qualifier");
            }
        }
    }
    //解析。
    public static Version parseVersion(String version) {
        if (version == null) {
            return emptyVersion;
        }

        version = version.trim();
        if (version.length() == 0) {
            return emptyVersion;
        }

        return new Version(version);
    }

    public int getMajor() {
        return major;
    }
    public int getMinor() {
        return minor;
    }
    public int getMicro() {
        return micro;
    }
    public String getQualifier() {
        return qualifier;
    }
```

```
public String toString() {
    String base = major + SEPARATOR + minor + SEPARATOR + micro;
    if (qualifier.length() == 0) {
        return base;
    }
    else {
        return base + SEPARATOR + qualifier;
    }
}
public int hashCode() {
    return (major << 24) + (minor << 16) + (micro << 8)
            + qualifier.hashCode();
}

public boolean equals(Object object) {
    if (object == this) {
        return true;
    }

    if (!(object instanceof Version)) {
        return false;
    }

    Version other = (Version) object;
    return (major == other.major) && (minor == other.minor)
            && (micro == other.micro)
            && qualifier.equals(other.qualifier);
}
//比较，前3部分数字，最后一部分是字母。
public int compareTo(Object object) {
```

```
        if (object == this) {
            return 0;
        }

        Version other = (Version) object;
        int result = major - other.major;
        if (result != 0) {
            return result;
        }
        result = minor - other.minor;
        if (result != 0) {
            return result;
        }
        result = micro - other.micro;
        if (result != 0) {
            return result;
        }
        return qualifier.compareTo(other.qualifier);
    }
}
```

1.4.8 IConfigurable

支持可配置服务的 Bundle 使用，用于测试服务对象是否实现 IConfigurable，已被废除。

```
package org.osgi.framework;

public interface Configurable {
    public Object getConfigurationObject();
}
```

2 osgi.eclipse.framework.launch

2.1 Launch

这个类提供了运行 OSGi 框架的入口。它根据命令参数配置 OSGi 框架。它支持的参数有：

（1）-con[sole]:[:port]——表示使用一个控制台窗口启动 OSGi 框架。如果指定了 Port 了，则控制台将从指定端口获取命令，否则佛那个 System.in 和 System.out。

（2）-adaptor[:adaptor-name] [adaptor-args]——指定使用的 FrameworkApdater 的实现类。

（3）-app[lication]:application-args——指定启动时间指定应用系统参数。

如果没有指定参数，则默认是使用默认 FrameworkAdapter、没有控制台、没有远程 Agent。

```
package org.eclipse.osgi.framework.launcher;


import java.lang.reflect.Constructor;
import java.util.*;
import org.eclipse.osgi.framework.adaptor.FrameworkAdaptor;
import org.eclipse.osgi.framework.internal.core.*;
import org.eclipse.osgi.util.NLS;


public class Launcher {


    protected String consolePort = "";
    protected boolean console = false;
    protected String adaptorClassName =
        "org.eclipse.osgi.baseadaptor.BaseAdaptor";
    protected final String osgiConsoleClazz
        = "org.eclipse.osgi.framework.internal.core.FrameworkConsole";
```

```
    protected String[] adaptorArgs = null;
    private static final String OSGI_CONSOLE_COMPONENT_NAME = "OSGi
Console";
    private static final String OSGI_CONSOLE_COMPONENT = "console.jar";

    //启动OSGi框架。
    public static void main(String args[]) {

        new Launcher().doIt(args);

    }

    public Launcher() {

    }

    protected void doIt(String[] args) {
        if (FrameworkProperties.getProperty(
            Constants.OSGI_COMPATIBILITY_BOOTDELEGATION) == null)
            FrameworkProperties.setProperty(
                Constants.OSGI_COMPATIBILITY_BOOTDELEGATION, "false");

        String[] consoleArgs = parseArgs(args);
        //创佳FrameworkAdapter。
        FrameworkAdaptor adaptor = null;
        try {
            adaptor = doAdaptor();
        } catch (Exception e) {
            System.out.println(Msg.LAUNCHER_ADAPTOR_ERROR);
            e.printStackTrace();
```

```
            return;
        }
        //创建OSGi，并启动。
        OSGi osgi = doOSGi(adaptor);
        if (osgi != null) {
            if (console) {
                doConsole(osgi, consoleArgs);
            } else {
                osgi.launch();
            }
        }
    }
    //解析OSGi启动参数。
    protected String[] parseArgs(String[] args) {
        Vector consoleArgsVector = new Vector();
        for (int i = 0; i < args.length; i++) {
            boolean match = false;

            String fullarg = args[i];
            int quoteidx = fullarg.indexOf("\"");
            if (quoteidx > 0) {
                if (quoteidx == fullarg.lastIndexOf("\"")) {
                    boolean stillparsing = true;
                    i++;
                    while (i < args.length && stillparsing) {
                        fullarg = fullarg + " " + args[i];
                        i++;
                        if (quoteidx < fullarg.lastIndexOf("\"")) {
                            stillparsing = false;
                        }
```

```
            }
        }
    } else {
        quoteidx = fullarg.indexOf("'");
        if (quoteidx > 0) {
            if (quoteidx == fullarg.lastIndexOf("'")) {
                boolean stillparsing = true;
                i++;
                while (i < args.length && stillparsing) {
                    fullarg = fullarg + " " + args[i];
                    i++;
                    if (quoteidx < fullarg.lastIndexOf("'")) {
                        stillparsing = false;
                    }
                }
            }
            fullarg = fullarg.replace('\'', '\"');
        }
    }

    Tokenizer tok = new Tokenizer(fullarg);
    if (tok.hasMoreTokens()) {
        String command = tok.getString(" ");
        StringTokenizer subtok = new StringTokenizer(
            command, ":");
        String subcommand = subtok.nextToken().toLowerCase();

        if (matchCommand("-console", subcommand, 4)) {
            _console(command);
            match = true;
```

```
            }
            if (matchCommand("-adaptor", subcommand, 2)) {
                _adaptor(command);
                match = true;
            }
            if (match == false) {
                consoleArgsVector.addElement(fullarg);
            }
        }
    }
    String[] consoleArgsArray =
        new String[consoleArgsVector.size()];
    Enumeration e = consoleArgsVector.elements();
    for (int i = 0; i < consoleArgsArray.length; i++) {
        consoleArgsArray[i] = (String) e.nextElement();
    }
    return consoleArgsArray;
}

public static boolean matchCommand(
    String command, String input, int minLength) {
    if (minLength <= 0) {
        minLength = command.length();
    }

    int length = input.length();

    if (minLength > length) {
        length = minLength;
    }
```

```
    return (command.regionMatches(0, input, 0, length));
}
//解析Console命令。
protected void _console(String command) {
    console = true;
    StringTokenizer tok = new StringTokenizer(command, ":");

    tok.nextToken();
    if (tok.hasMoreTokens()) {
        consolePort = tok.nextToken();
    }
}
//解析Adapter命令。
protected void _adaptor(String command) {
    Tokenizer tok = new Tokenizer(command);
    tok.getToken(":");
    tok.getChar();
    String adp = tok.getToken(":");
    if (adp != null && adp.length() > 0) {
        adaptorClassName = adp;
    }

    Vector v = new Vector();
    parseloop: while (true) {
        tok.getChar();
        String arg = tok.getString(":");
        if (arg == null)
            break parseloop;
        v.addElement(arg);
```

```
        }

        if (v != null) {
            int numArgs = v.size();
            adaptorArgs = new String[numArgs];
            Enumeration e = v.elements();
            for (int i = 0; i < numArgs; i++) {
                adaptorArgs[i] = (String) e.nextElement();
            }
        }
    }

    //初始化一个FrameworkAdapter。
    protected FrameworkAdaptor doAdaptor() throws Exception {
        Class adaptorClass = Class.forName(adaptorClassName);
        Class[] constructorArgs = new Class[] {String[].class};
        Constructor constructor = adaptorClass.getConstructor(
            constructorArgs);
        return (FrameworkAdaptor) constructor.newInstance(new Object[]
{adaptorArgs});
    }

    //创建一个OSGi对象。
    private OSGi doOSGi(FrameworkAdaptor adaptor) {
        return new OSGi(adaptor);
    }

    //启动一个线程来运行控制台。
    private void doConsole(OSGi osgi, String[] consoleArgs) {
```

```
Constructor consoleConstructor;
Object osgiconsole;
Class[] parameterTypes;
Object[] parameters;

try {
    Class osgiconsoleClass = Class.forName(osgiConsoleClazz);
    if (consolePort.length() == 0) {
        parameterTypes = new Class[] {OSGi.class, String[].class};
        parameters = new Object[] {osgi, consoleArgs};
    } else {
        parameterTypes = new Class[] {
            OSGi.class, int.class, String[].class};
        parameters = new Object[] {
            osgi, new Integer(consolePort), consoleArgs};
    }
    consoleConstructor = osgiconsoleClass.getConstructor(
        parameterTypes);
    osgiconsole = consoleConstructor.newInstance(parameters);

    Thread t = new Thread(
        ((Runnable) osgiconsole), OSGI_CONSOLE_COMPONENT_NAME);
    t.start();
} catch (NumberFormatException nfe) {
    System.err.println(
        NLS.bind(Msg.LAUNCHER_INVALID_PORT, consolePort));
} catch (Exception ex) {
    informAboutMissingComponent(
        OSGI_CONSOLE_COMPONENT_NAME, OSGI_CONSOLE_COMPONENT);
}
```

```
    }
    //通知丢失一个组件的消息。
    void informAboutMissingComponent(String component, String jar) {
        System.out.println();
        System.out.print(NLS.bind(
            Msg.LAUNCHER_COMPONENT_MISSING, component));
        System.out.println(NLS.bind(Msg.LAUNCHER_COMPONENT_JAR, jar));
        System.out.println();
    }
}
```

3 osgi.eclipse.framework.internal

3.1 osgi.eclipse.framework.internal.core

3.1.1 OSGi

OSGi 主类。这个用于启动框架。

```
package org.eclipse.osgi.framework.internal.core;

import org.eclipse.osgi.framework.adaptor.FrameworkAdaptor;

public class OSGi {
    //框架。
    protected Framework framework;
    //框架适配器。
    public OSGi(FrameworkAdaptor adaptor) {
        framework = createFramework(adaptor);
    }
```

```
    //销毁OSGi框架，释放框架资源。
    public void close() {
        framework.close();
    }
    public void launch() {
        framework.launch();
    }
    public void shutdown() {
        framework.shutdown();
    }
    public boolean isActive() {
        return (framework.isActive());
    }
    public org.osgi.framework.BundleContext getBundleContext() {
        return (framework.systemBundle.getContext());
    }
    protected Framework createFramework(FrameworkAdaptor adaptor) {
        return (new Framework(adaptor));
    }
    protected void displayBanner() {
        System.out.println();
        System.out.print(Msg.ECLIPSE_OSGI_NAME);
        System.out.print(" ");
        System.out.println(Msg.ECLIPSE_OSGI_VERSION);
        System.out.println();
        System.out.println(Msg.OSGI_VERSION);
        System.out.println();
        System.out.println(Msg.ECLIPSE_COPYRIGHT);
    }
}
```

3.1.2 Framework

核心 OSGi 框架类。

```
package org.eclipse.osgi.framework.internal.core;


import java.io.*;
import java.lang.reflect.*;
import java.net.*;
import java.security.*;
import java.util.*;
import org.eclipse.core.runtime.internal.adaptor.ContextFinder;
import org.eclipse.osgi.framework.adaptor.*;
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.framework.eventmgr.*;
import org.eclipse.osgi.framework.internal.protocol.
       ContentHandlerFactory;
import org.eclipse.osgi.framework.internal.protocol.
       StreamHandlerFactory;
import org.eclipse.osgi.framework.log.FrameworkLog;
import org.eclipse.osgi.framework.log.FrameworkLogEntry;
import org.eclipse.osgi.framework.util.SecureAction;
import org.eclipse.osgi.internal.profile.Profile;
import org.eclipse.osgi.util.ManifestElement;
import org.eclipse.osgi.util.NLS;
import org.osgi.framework.*;


public class Framework implements EventDispatcher, EventPublisher {
    //用于设置上下文类型加载器父加载器类型的系统属性。
    private static final String PROP_CONTEXTCLASSLOADER_PARENT =
        "osgi.contextClassLoaderParent";
```

```
private static final String CONTEXTCLASSLOADER_PARENT_APP = "app";
private static final String CONTEXTCLASSLOADER_PARENT_EXT = "ext";
private static final String CONTEXTCLASSLOADER_PARENT_BOOT = "boot";
private static final String CONTEXTCLASSLOADER_PARENT_FWK = "fwk";

private static String J2SE = "J2SE-";
private static String JAVASE = "JavaSE-";
private static String PROFILE_EXT = ".profile";
//框架适配器。
protected FrameworkAdaptor adaptor;
//框架属性集合。
protected Properties properties;
//框架是否启动。
protected boolean active;
//Bundle仓库。
protected BundleRepository bundles;
//包管理。
protected PackageAdminImpl packageAdmin;
//权限管理。
protected PermissionAdminImpl permissionAdmin;
//启动级别管理器。
protected StartLevelManager startLevelManager;
//服务注册表。
protected ServiceRegistry serviceRegistry;
//下一个服务ID。
protected long serviceid;
//事件。
protected EventListeners bundleEvent;
protected static final int BUNDLEEVENT = 1;
protected EventListeners bundleEventSync;
```

```
protected static final int BUNDLEEVENTSYNC = 2;
protected EventListeners serviceEvent;
protected static final int SERVICEEVENT = 3;
protected EventListeners frameworkEvent;
protected static final int FRAMEWORKEVENT = 4;
protected static final int BATCHEVENT_BEGIN = Integer.MIN_VALUE + 1;
protected static final int BATCHEVENT_END = Integer.MIN_VALUE;
protected EventManager eventManager;
//同步安装。
protected Hashtable installLock;
//系统Bundle。
protected SystemBundle systemBundle;
//启动代理。
String[] bootDelegation;
String[] bootDelegationStems;
boolean bootDelegateAll = false;
boolean contextBootDelegation = "true".equals(
    FrameworkProperties.getProperty(
    "osgi.context.bootdelegation", "true"));
boolean compatibiltyBootDelegation = false;



protected static AliasMapper aliasMapper = new AliasMapper();
protected ConditionalPermissionAdminImpl condPermAdmin;
SecureAction secureAction =
    (SecureAction) AccessController.doPrivileged(
        SecureAction.createSecureAction());
private HashMap adminPermissions = new HashMap();


private StreamHandlerFactory streamHandlerFactory;
```

```
private ContentHandlerFactory contentHandlerFactory;


static {
    Class c;
    c = GetDataFileAction.class;
    c.getName();
}


static class GetDataFileAction implements PrivilegedAction {
    private AbstractBundle bundle;
    private String filename;

    public GetDataFileAction(
        AbstractBundle bundle, String filename)
    {
        this.bundle = bundle;
        this.filename= filename;
    }

    public Object run() {
        return bundle.getBundleData().getDataFile(filename);
    }
}


//使用一个适配器创建一个Framewok。
public Framework(FrameworkAdaptor adaptor) {
    initialize(adaptor);
}
//初始化框架到unlaunched状态。
protected void initialize(FrameworkAdaptor adaptor) {
```

```
long start = System.currentTimeMillis();
//设置适配器。
this.adaptor = adaptor;
active = false;
installSecurityManager();

setNLSFrameworkLog();
initializeContextFinder();
//初始化适配器。
adaptor.initialize(this);
try {
    //初始化存储器。
    adaptor.initializeStorage();
} catch (IOException e)  //致命错误。
{
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
}
//使用适配器初始化属性。
initializeProperties(adaptor.getProperties());
packageAdmin = new PackageAdminImpl(this);
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    try {
        permissionAdmin = new PermissionAdminImpl(
            this, adaptor.getPermissionStorage());
    } catch (IOException e) //致命错误。
    {
        e.printStackTrace();
        throw new RuntimeException(e.getMessage());
```

```
    }
    try {
        condPermAdmin = new ConditionalPermissionAdminImpl(
            this, adaptor.getPermissionStorage());
    } catch (IOException e) //致命错误。
    {
        e.printStackTrace();
        throw new RuntimeException(e.getMessage());
    }
}
//启动级别管理器。
startLevelManager = new StartLevelManager(this);
//事件初始化。
eventManager = new EventManager("Framework Event Dispatcher");
bundleEvent = new EventListeners();
bundleEventSync = new EventListeners();
serviceEvent = new EventListeners();
frameworkEvent = new EventListeners();
//创建服务注册表。
serviceid = 1;
serviceRegistry = adaptor.getServiceRegistry();
//初始化安装锁。
installLock = new Hashtable(10);
//创建系统Bundle。
createSystemBundle();
loadVMProfile();
setBootDelegation();
installURLStreamHandlerFactory(systemBundle.context, adaptor);
installContentHandlerFactory(systemBundle.context, adaptor);
//为所有安装的Bundle创建Bundle数据对象。
```

```
        BundleData[] bundleDatas = adaptor.getInstalledBundles();
        //初始化Bundle仓库。
        bundles = new BundleRepository(
            bundleDatas == null ? 10 : bundleDatas.length + 1);
        //添加系统Bundle。
        bundles.add(systemBundle);
        //为每一个Bundle数据创建Bundle，并添加到Bundle仓库中。
        if (bundleDatas != null) {
            for (int i = 0; i < bundleDatas.length; i++) {
                try {
                    AbstractBundle bundle =
                        AbstractBundle.createBundle(
                        bundleDatas[i], this);
                    bundles.add(bundle);
                } catch (BundleException be) {
                    publishFrameworkEvent(
                        FrameworkEvent.ERROR, systemBundle, be);
                }
            }
        }
        //初始化完毕。
    }

    private void setNLSFrameworkLog() {
        try {
            Field frameworkLogField = NLS.class.getDeclaredField(
                "frameworkLog");
            frameworkLogField.setAccessible(true);
            frameworkLogField.set(null, adaptor.getFrameworkLog());
        } catch (Exception e) {
```

```
        }
    }
    //创建系统Bundle。
    private void createSystemBundle() {
        try {
            systemBundle = new SystemBundle(this);
        } catch (BundleException e) { //致命错误。
            e.printStackTrace();
            throw new RuntimeException(
                NLS.bind(Msg.OSGI_SYSTEMBUNDLE_CREATE_EXCEPTION,
                e.getMessage()));
        }
    }
    protected void initializeProperties(Properties adaptorProperties) {
        properties = FrameworkProperties.getProperties();
        Enumeration enumKeys = adaptorProperties.propertyNames();
        while (enumKeys.hasMoreElements()) {
            String key = (String) enumKeys.nextElement();
            if (properties.getProperty(key) == null) {
                properties.put(
                    key, adaptorProperties.getProperty(key));
            }
        }
        properties.put(Constants.FRAMEWORK_VENDOR,
            Constants.OSGI_FRAMEWORK_VENDOR);
        properties.put(Constants.FRAMEWORK_VERSION,
            Constants.OSGI_FRAMEWORK_VERSION);
        String value = properties.getProperty(
            Constants.FRAMEWORK_PROCESSOR);
        if (value == null) {
```

```
            value = properties.getProperty(Constants.JVM_OS_ARCH);
            if (value != null) {
                properties.put(Constants.FRAMEWORK_PROCESSOR, value);
            }
        }
        value = properties.getProperty(Constants.FRAMEWORK_OS_NAME);
        if (value == null) {
            value = properties.getProperty(Constants.JVM_OS_NAME);
            try {
                String canonicalValue = (String) aliasMapper.aliasOSName(
                    value);
                if (canonicalValue != null) {
                    value = canonicalValue;
                }
            } catch (ClassCastException ex) {
            }
            if (value != null) {
                properties.put(Constants.FRAMEWORK_OS_NAME, value);
            }
        }
        value = properties.getProperty(Constants.FRAMEWORK_OS_VERSION);
        if (value == null) {
            value = properties.getProperty(Constants.JVM_OS_VERSION);
            if (value != null) {
                int space = value.indexOf(' ');
                if (space > 0) {
                    value = value.substring(0, space);
                }
                properties.put(Constants.FRAMEWORK_OS_VERSION, value);
            }
```

```
        }
        value = properties.getProperty(Constants.FRAMEWORK_LANGUAGE);
        if (value == null)
            properties.put(Constants.FRAMEWORK_LANGUAGE,
                Locale.getDefault().getLanguage());
        properties.put(Constants.SUPPORTS_FRAMEWORK_FRAGMENT, "true");
        properties.put(
            Constants.SUPPORTS_FRAMEWORK_REQUIREBUNDLE, "true");
    }
    //设置启动代理。
    private void setBootDelegation() {
        compatibiltyBootDelegation = "true".equals(
            FrameworkProperties.getProperty(
            Constants.OSGI_COMPATIBILITY_BOOTDELEGATION));
        String bootDelegationProp = properties.getProperty(
            Constants.OSGI_BOOTDELEGATION);
        if (bootDelegationProp == null)
            return;
        if (bootDelegationProp.trim().length() == 0)
            return;
        String[] bootPackages = ManifestElement.getArrayFromList(
            bootDelegationProp);
        ArrayList exactMatch = new ArrayList(bootPackages.length);
        ArrayList stemMatch = new ArrayList(bootPackages.length);
        for (int i = 0; i < bootPackages.length; i++) {
            if (bootPackages[i].equals("*")) {
                bootDelegateAll = true;
                return;
            } else if (bootPackages[i].endsWith("*")) {
                if (bootPackages[i].length() > 2 &&
```

```
                bootPackages[i].endsWith(".*"))
                stemMatch.add(bootPackages[i].substring(0,
                    bootPackages[i].length() - 1));
            } else {
                exactMatch.add(bootPackages[i]);
            }
        }
        if (exactMatch.size() > 0)
            bootDelegation = (String[]) exactMatch.toArray(
            new String[exactMatch.size()]);
        if (stemMatch.size() > 0)
            bootDelegationStems = (String[]) stemMatch.toArray(
            new String[stemMatch.size()]);
    }
    //装载虚拟机配置。
    private void loadVMProfile() {
        Properties profileProps = findVMProfile();
        String systemExports = properties.getProperty(
            Constants.OSGI_FRAMEWORK_SYSTEM_PACKAGES);
        if (systemExports == null) {
            systemExports = profileProps.getProperty(
                Constants.OSGI_FRAMEWORK_SYSTEM_PACKAGES);
            if (systemExports != null)
                properties.put(
                    Constants.OSGI_FRAMEWORK_SYSTEM_PACKAGES,
                    systemExports);
        }

        String type = properties.getProperty(
            Constants.OSGI_JAVA_PROFILE_BOOTDELEGATION);
```

```
    String profileBootDelegation =
        profileProps.getProperty(Constants.OSGI_BOOTDELEGATION);
    if (Constants.OSGI_BOOTDELEGATION_OVERRIDE.equals(type)) {
        if (profileBootDelegation == null)
            properties.remove(Constants.OSGI_BOOTDELEGATION);
    else
            properties.put(Constants.OSGI_BOOTDELEGATION,
                profileBootDelegation);
    } else if (Constants.OSGI_BOOTDELEGATION_NONE.equals(type))
        properties.remove(Constants.OSGI_BOOTDELEGATION);
    if (properties.getProperty(
        Constants.FRAMEWORK_EXECUTIONENVIRONMENT) == null) {
        String ee = profileProps.getProperty(
            Constants.FRAMEWORK_EXECUTIONENVIRONMENT,
            profileProps.getProperty
                (Constants.OSGI_JAVA_PROFILE_NAME));
        if (ee != null)
            properties.put(
                Constants.FRAMEWORK_EXECUTIONENVIRONMENT, ee);
    }
}

private Properties findVMProfile() {
    ……
}

private URL findNextBestProfile(
    String javaEdition, Version javaVersion) {
    URL result = null;
    int minor = javaVersion.getMinor();
```

```
    do {
        result = findInSystemBundle(
            javaEdition + javaVersion.getMajor() + "." + minor +
            PROFILE_EXT);
        minor = minor - 1;
    } while (result == null && minor > 0);
    return result;
}

private URL findInSystemBundle(String entry) {
    URL result = systemBundle.getEntry(entry);
    if (result == null) {
        ClassLoader loader=getClass().getClassLoader();
        result = loader==null ?
            ClassLoader.getSystemResource(entry) :
            loader.getResource(entry);
    }
    return result;
}

protected boolean isActive() {
    return (active);
}
//管理OSGi框架。
public synchronized void close() {
    //关闭框架。
    if (active) {
        shutdown();
    }
    //关闭Bundle仓库。
```

```
synchronized (bundles) {
    List allBundles = bundles.getBundles();
    int size = allBundles.size();
    for (int i = 0; i < size; i++) {
        AbstractBundle bundle =
            (AbstractBundle) allBundles.get(i);
        bundle.close();
    }
    bundles.removeAllBundles();
}
serviceRegistry = null;
//移除监听器。
if (bundleEvent != null) {
    bundleEvent.removeAllListeners();
    bundleEvent = null;
}
if (bundleEventSync != null) {
    bundleEventSync.removeAllListeners();
    bundleEventSync = null;
}
if (serviceEvent != null) {
    serviceEvent.removeAllListeners();
    serviceEvent = null;
}
if (frameworkEvent != null) {
    frameworkEvent.removeAllListeners();
    frameworkEvent = null;
}
if (eventManager != null) {
    eventManager.close();
```

```
            eventManager = null;
        }
        //释放其它资源。
        permissionAdmin = null;
        condPermAdmin = null;
        packageAdmin = null;
        adaptor = null;
        uninstallURLStreamHandlerFactory();
        uninstallContentHandlerFactory();
    }
    //启动框架。
    public synchronized void launch() {
        if (active) {
            return;
        }
        active = true;
        //回复系统Bundle。
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            Debug.println("Trying to launch framework");
        }
        systemBundle.resume();
    }
    //停止框架。
    public synchronized void shutdown() {
        if (!active) {
            return;
        }
        //设置系统Bundle状态，发布事件。
        systemBundle.state = AbstractBundle.STOPPING;
        publishBundleEvent(BundleEvent.STOPPING, systemBundle);
```

```
    //开始停止。
    try {
        adaptor.frameworkStopping(systemBundle.getContext());
    } catch (Throwable t) {
        publishFrameworkEvent(FrameworkEvent.ERROR,
            systemBundle, t);
    }
    //挂起系统Bundle。
    if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
        Debug.println("Trying to shutdown Framework");
    }
    systemBundle.suspend();
    try {
        adaptor.compactStorage();
    } catch (IOException e) {
        publishFrameworkEvent(FrameworkEvent.ERROR,
            systemBundle, e);
    }
    //标记框架停止。
    active = false;
}

//创建并验证一个Bundle。
AbstractBundle createAndVerifyBundle(BundleData bundledata)
    throws BundleException {
    //验证Bundle。
    if (bundledata.getSymbolicName() != null) {
        AbstractBundle installedBundle = getBundleBySymbolicName(
            bundledata.getSymbolicName(), bundledata.getVersion());
        if (installedBundle != null &&
```

```
                installedBundle.getBundleId() !=
                    bundledata.getBundleID()) {
                throw new BundleException(NLS.bind(
                    Msg.BUNDLE_INSTALL_SAME_UNIQUEID,
                    new Object[] {installedBundle.getSymbolicName(),
                    installedBundle.getVersion().toString(),
                    installedBundle.getLocation()}));
            }
        }
        //验证执行环境。
        verifyExecutionEnvironment(bundledata.getManifest());
        //创建Bundle。
        return AbstractBundle.createBundle(bundledata, this);
    }
    //验证执行环境。
    protected boolean verifyExecutionEnvironment(Dictionary manifest)
        throws BundleException {
        if (!Boolean.valueOf(FrameworkProperties.getProperty(
            Constants.ECLIPSE_EE_INSTALL_VERIFY,
            Boolean.TRUE.toString())).booleanValue())
            return true;
        String headerValue = (String) manifest.get(
            Constants.BUNDLE_REQUIREDEXECUTIONENVIRONMENT);
        if (headerValue == null) {
            return true;
        }
        ManifestElement[] bundleRequiredEE =
            ManifestElement.parseHeader(
            Constants.BUNDLE_REQUIREDEXECUTIONENVIRONMENT,
            headerValue);
```

```
        if (bundleRequiredEE.length == 0) {
            return true;
        }
        String systemEE = FrameworkProperties.getProperty(
            Constants.FRAMEWORK_EXECUTIONENVIRONMENT);
        if (systemEE != null && !systemEE.equals("")) {
            ManifestElement[] systemEEs = ManifestElement.parseHeader(
                Constants.BUNDLE_REQUIREDEXECUTIONENVIRONMENT,
                systemEE);
            for (int i = 0; i < systemEEs.length; i++) {
                for (int j = 0; j < bundleRequiredEE.length; j++) {
                    if (systemEEs[i].getValue().equals(
                        bundleRequiredEE[j].getValue())) {
                        return true;
                    }
                }
            }
        }
        StringBuffer bundleEE = new StringBuffer(25);
        for (int i = 0; i < bundleRequiredEE.length; i++) {
            if (i > 0) {
                bundleEE.append(",");
            }
            bundleEE.append(bundleRequiredEE[i].getValue());
        }
        throw new BundleException(
            NLS.bind(Msg.BUNDLE_INSTALL_REQUIRED_EE_EXCEPTION,
            bundleEE.toString()));
    }
```

```
public String getProperty(String key) {
    return properties.getProperty(key);
}
protected String getProperty(String key, String def) {
    return properties.getProperty(key, def);
}
protected Object setProperty(String key, String value) {
    return properties.put(key, value);
}
//从Bundle位置安装一个Bundle。
public AbstractBundle installBundle(final String location)
    throws BundleException {
    final AccessControlContext callerContext =
        AccessController.getContext();
    return installWorker(location, new PrivilegedExceptionAction()
     {
        public Object run() throws BundleException {
            URLConnection source =
                adaptor.mapLocationToURLConnection(location);
            return installWorkerPrivileged(
                location, source, callerContext);
        }
    });
}
protected AbstractBundle installBundle(final String location,
    final InputStream in) throws BundleException {
    final AccessControlContext callerContext =
        AccessController.getContext();
    return installWorker(location, new PrivilegedExceptionAction() {
        public Object run() throws BundleException {
```

```
                URLConnection source = new BundleSource(in);
                return installWorkerPrivileged(
                    location, source, callerContext);
            }
        });
    }
    //安装一个Bundle的方法。
    protected AbstractBundle installWorker(String location,
        PrivilegedExceptionAction action) throws BundleException {
        //加锁。
        synchronized (installLock) {
            while (true) {
                //判断Bundle是否已经存在，如果存在直接返回。
                AbstractBundle bundle = getBundleByLocation(location);
                if (bundle != null) {
                    return bundle;
                }
                Thread current = Thread.currentThread();
                //检查是否有现成申请保护。
                Thread reservation = (Thread) installLock.put(
                    location, current);
                if (reservation == null) { //没有保护。
                    break;
                }
                //如果保护是当前线程，则抛出重复安装异常。
                if (current.equals(reservation)) {
                    throw new BundleException(
                        Msg.BUNDLE_INSTALL_RECURSION_EXCEPTION);
                }
                try {
```

```
                //等待释放锁。
                installLock.wait();
            } catch (InterruptedException e) {
            }
        }
    }

    try {
        //使用一个PrivilegedExceptionAction创建Bundle。
        //它将调用installWorkerPrivileged。
        AbstractBundle bundle =
            (AbstractBundle) AccessController.doPrivileged(action);
        publishBundleEvent(BundleEvent.INSTALLED, bundle);
        return bundle;
    } catch (PrivilegedActionException e) {
        if (e.getException() instanceof RuntimeException)
            throw (RuntimeException) e.getException();
        throw (BundleException) e.getException();
    } finally {
        synchronized (installLock) {
            //释放。
            installLock.remove(location);
            //通知。
            installLock.notifyAll();
        }
    }
}
//调用FrameworkAdapter在持久存储安装一个Bundle。
protected AbstractBundle installWorkerPrivileged(String location,
    URLConnection source, AccessControlContext callerContext)
```

```
throws BundleException {
//获取BundleOperation。
BundleOperation storage = adaptor.installBundle(
    location, source);
final AbstractBundle bundle;
try {
    //获取BundleData。
    BundleData bundledata = storage.begin();
    //验证BundleData。
    bundle = createAndVerifyBundle(bundledata);
    if (Debug.DEBUG) {
        BundleWatcher bundleStats = adaptor.getBundleWatcher();
        if (bundleStats != null)
            bundleStats.watchBundle(
            bundle, BundleWatcher.START_INSTALLING);
    }
    try {
        String[] nativepaths = selectNativeCode(bundle);
        if (nativepaths != null) {
            bundledata.installNativeCode(nativepaths);
        }
        //装载Bundel。
        bundle.load();
        //权限检查。
        if (System.getSecurityManager() != null) {
            final boolean extension = (bundledata.getType() &
                (BundleData.TYPE_BOOTCLASSPATH_EXTENSION |
                BundleData.TYPE_FRAMEWORK_EXTENSION)) != 0;
            if (extension && !bundle.hasPermission(
                new AllPermission()))
```

```
                throw new BundleException(
                    Msg.BUNDLE_EXTENSION_PERMISSION,
                    new SecurityException(
                        Msg.BUNDLE_EXTENSION_PERMISSION));
            try {
                AccessController.doPrivileged(
                    new PrivilegedExceptionAction() {
                    public Object run() throws Exception {
                        checkAdminPermission(bundle,
                             AdminPermission.LIFECYCLE);
                        if (extension)
                            checkAdminPermission(bundle,
                            AdminPermission.EXTENSIONLIFECYCLE);
                        return null;
                    }
                }, callerContext);
            } catch (PrivilegedActionException e) {
                throw e.getException();
            }
        }
        storage.commit(false);
    } catch (Throwable error) {
        synchronized (bundles) {
            bundle.unload();
        }
        bundle.close();
        throw error;
    } finally {
        if (Debug.DEBUG) {
            BundleWatcher bundleStats =
```

```
                    adaptor.getBundleWatcher();
                if (bundleStats != null)
                    bundleStats.watchBundle(
                        bundle, BundleWatcher.END_INSTALLING);
            }
        }
        //Bundle成功安装。
        bundles.add(bundle);
    } catch (Throwable t) {//撤销安装，发布事件。
        try {
            storage.undo();
        } catch (BundleException ee) {
            publishFrameworkEvent(
                FrameworkEvent.ERROR, systemBundle, ee);
        }
        if (t instanceof SecurityException)
            throw (SecurityException) t;
        if (t instanceof BundleException)
            throw (BundleException) t;
        throw new BundleException(t.getMessage(), t);
    }
    return bundle;
}

String[] selectNativeCode(org.osgi.framework.Bundle bundle)
    throws BundleException {
    ……
}

private boolean isBncGreaterThan(BundleNativeCode candidate,
```

```
     BundleNativeCode highestRanking,
    Version version, String language) {
    Version currentHigh = highestRanking.matchOSVersion(version);
    Version candidateHigh = candidate.matchOSVersion(version);
    if (currentHigh.compareTo(candidateHigh) < 0)
        return true;
    if (highestRanking.matchLanguage(language) <
        candidate.matchLanguage(language))
        return true;
    return false;
}
//获取指定ID的Bundle。
public AbstractBundle getBundle(long id) {
    synchronized (bundles) {
        return bundles.getBundle(id);
    }
}
//通过特征名称、版本号获取Bundle。
public AbstractBundle getBundleBySymbolicName(
    String symbolicName, Version version) {
    synchronized (bundles) {
        return bundles.getBundle(symbolicName, version);
    }
}
//获取所有Bundle。
protected BundleRepository getBundles() {
    return (bundles);
}
protected AbstractBundle[] getAllBundles() {
    synchronized (bundles) {
```

```
            List allBundles = bundles.getBundles();
            int size = allBundles.size();
            if (size == 0) {
                return (null);
            }
            AbstractBundle[] bundlelist = new AbstractBundle[size];
            allBundles.toArray(bundlelist);
            return (bundlelist);
        }
    }
    //恢复一个Bundle。
    protected void resumeBundle(AbstractBundle bundle) {
        if (bundle.isActive()) {
            return;
        }
        try {
            bundle.resume();
        } catch (BundleException be) {
            publishFrameworkEvent(FrameworkEvent.ERROR, bundle, be);
        }
    }
    //挂起一个Bundle。
    protected boolean suspendBundle(AbstractBundle bundle, boolean lock)
    {
        boolean changed = false;
        if (!bundle.isActive() || bundle.isFragment()) {
            return changed;
        }
        try {
            bundle.suspend(lock);
```

```
        } catch (BundleException be) {
            publishFrameworkEvent(FrameworkEvent.ERROR, bundle, be);
        }
        if (!bundle.isActive()) {
            changed = true;
        }
        return (changed);
    }
    //获取指定Location的Bundle。
    protected AbstractBundle getBundleByLocation(String location) {
        synchronized (bundles) {
            final String finalLocation = location;

            return (AbstractBundle) AccessController.doPrivileged(
                new PrivilegedAction() {
                public Object run() {
                    List allBundles = bundles.getBundles();
                    int size = allBundles.size();
                    for (int i = 0; i < size; i++) {
                        AbstractBundle bundle =
                            (AbstractBundle) allBundles.get(i);
                        if (finalLocation.equals(bundle.getLocation())) {
                            return (bundle);
                        }
                    }
                    return (null);
                }
            });
        }
    }
```

```
//按特征名称获取Bundle。
protected AbstractBundle[] getBundleBySymbolicName(
    String symbolicName) {
    synchronized (bundles) {
        return bundles.getBundles(symbolicName);
    }
}
//获取服务引用。
protected ServiceReference[] getServiceReferences(String clazz,
    String filterstring, BundleContextImpl context,
    boolean allservices) throws InvalidSyntaxException {
    FilterImpl filter = (filterstring == null) ?
        null : new FilterImpl(filterstring);
    ServiceReference[] services = null;
    if (clazz != null) {
        try { //测试访问权限。
            checkGetServicePermission(clazz);
        } catch (SecurityException se) {
            return (null);
        }
    }
    synchronized (serviceRegistry) {
        //查询服务引用。
        services = serviceRegistry.lookupServiceReferences(
            clazz, filter);
        if (services == null) {
            return null;
        }
        int removed = 0;
        for (int i = services.length - 1; i >= 0; i--) {
```

```
        ServiceReferenceImpl ref =
            (ServiceReferenceImpl) services[i];
        String[] classes = ref.getClasses();
        //接口兼容判断。
        if (allservices || context.isAssignableTo(
            (ServiceReferenceImpl) services[i])) {
            if (clazz == null)
                try {
                    checkGetServicePermission(classes);
                } catch (SecurityException se) {
                    services[i] = null;
                    removed++;
                }
        } else {
            services[i] = null;
            removed++;
        }
    }
    //删除不兼容的服务。
    if (removed > 0) {
        ServiceReference[] temp = services;
        services = new ServiceReference[temp.length - removed];
        for (int i = temp.length - 1; i >= 0; i--) {
            if (temp[i] == null)
                removed--;
            else
                services[i - removed] = temp[i];
        }
    }
```

```
    }
    return services == null || services.length == 0 ? null : services;
}

protected long getNextServiceId() {
    long id = serviceid;
    serviceid++;
    return (id);
}

protected File getDataFile(final AbstractBundle bundle,
    final String filename) {
    return (File) AccessController.doPrivileged(
        new GetDataFileAction(bundle, filename));
        }
//
//检查权限。
//

//传送框架事件。
public void publishFrameworkEvent(int type,
    org.osgi.framework.Bundle bundle, Throwable throwable) {
    if (frameworkEvent != null) {
        if (bundle == null)
            bundle = systemBundle;
        final FrameworkEvent event = new FrameworkEvent(
            type, bundle, throwable);
        if (System.getSecurityManager() == null) {
            publishFrameworkEventPrivileged(event);
        } else {
```

```
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    publishFrameworkEventPrivileged(event);
                    return null;
                }
            });
        }
    }
}
//发布框架事件。
public void publishFrameworkEventPrivileged(FrameworkEvent event) {
    if (event.getType() == FrameworkEvent.ERROR) {
        FrameworkLog frameworkLog = adaptor.getFrameworkLog();
        if (frameworkLog != null)
            frameworkLog.log(event);
    }

    ListenerQueue listeners = new ListenerQueue(eventManager);
    ListenerQueue contexts = new ListenerQueue(eventManager);
    synchronized (frameworkEvent) {
        //将BundleContext的监听器添加到队列。
        contexts.queueListeners(frameworkEvent, this);
        contexts.dispatchEventSynchronous(FRAMEWORKEVENT,
            listeners);
    }
    listeners.dispatchEventAsynchronous(FRAMEWORKEVENT, event);
}
//发布Bundle事件。
public void publishBundleEvent(int type,
    org.osgi.framework.Bundle bundle) {
```

```
    if ((bundleEventSync != null) || (bundleEvent != null)) {
        final BundleEvent event = new BundleEvent(type, bundle);
        if (System.getSecurityManager() == null) {
            publishBundleEventPrivileged(event);
        } else {
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    publishBundleEventPrivileged(event);
                    return null;
                }
            });
        }
    }
}
//验证后发布Bundle事件。
public void publishBundleEventPrivileged(BundleEvent event) {
    //获取同步和异步监听器。
    ListenerQueue listenersSync = null;
    if (bundleEventSync != null) {
        listenersSync = new ListenerQueue(eventManager);
        ListenerQueue contexts = new ListenerQueue(eventManager);
        synchronized (bundleEventSync) {
            contexts.queueListeners(bundleEventSync, this);
            contexts.dispatchEventSynchronous(
                BUNDLEEVENTSYNC, listenersSync);
        }
    }
    ListenerQueue listenersAsync = null;
    if (bundleEvent != null && (event.getType() &
        (BundleEvent.STARTING | BundleEvent.STOPPING |
```

```
            BundleEvent.LAZY_ACTIVATION)) == 0)  {
            listenersAsync = new ListenerQueue(eventManager);
            ListenerQueue contexts = new ListenerQueue(eventManager);
            synchronized (bundleEvent) {
                contexts.queueListeners(bundleEvent, this);
                contexts.dispatchEventSynchronous(
                    BUNDLEEVENT, listenersAsync);
            }
        }
        //派发事件。
        if (listenersSync != null) {
            listenersSync.dispatchEventSynchronous(
                BUNDLEEVENTSYNC, event);
        }
        if (listenersAsync != null) {
            listenersAsync.dispatchEventAsynchronous(
                BUNDLEEVENT, event);
        }
    }
    //发布服务事件。
    public void publishServiceEvent(
        int type, org.osgi.framework.ServiceReference reference) {
        if (serviceEvent != null) {
            final ServiceEvent event = new ServiceEvent(type, reference);
            if (System.getSecurityManager() == null) {
                publishServiceEventPrivileged(event);
            } else {
                AccessController.doPrivileged(new PrivilegedAction() {
                    public Object run() {
                        publishServiceEventPrivileged(event);
```

```
                    return null;
                }
            });
        }
    }
}
//权限验证通过后，真正开始发布事件。
public void publishServiceEventPrivileged(ServiceEvent event) {
    ListenerQueue listeners = new ListenerQueue(eventManager);
    ListenerQueue contexts = new ListenerQueue(eventManager);
    synchronized (serviceEvent) {
        contexts.queueListeners(serviceEvent, this);
        contexts.dispatchEventSynchronous(SERVICEEVENT, listeners);
    }
    listeners.dispatchEventSynchronous(SERVICEEVENT, event);
}
//框架顶层事件发布。
public void dispatchEvent(Object l, Object lo,
    int action, Object object) {
    try {
        BundleContextImpl context = (BundleContextImpl) l;
        if (context.isValid())//如果上下文可用。
        {
            ListenerQueue queue = (ListenerQueue) object;
            switch (action) {
                case BUNDLEEVENT : {
                    queue.queueListeners(
                        context.bundleEvent, context);
                    break;
                }
```

```
                case BUNDLEEVENTSYNC : {
                    queue.queueListeners(
                        context.bundleEventSync, context);
                    break;
                }
                case SERVICEEVENT : {
                    queue.queueListeners(
                        context.serviceEvent, context);
                    break;
                }
                case FRAMEWORKEVENT : {
                    queue.queueListeners(
                        context.frameworkEvent, context);
                    break;
                }
            }
        }
    } catch (Throwable t) {
        adaptor.handleRuntimeError(t);
        publisherror: {
            if (action == FRAMEWORKEVENT) {
                FrameworkEvent event = (FrameworkEvent) object;
                if (event.getType() == FrameworkEvent.ERROR) {
                    break publisherror;
                }
            }
            BundleContextImpl context = (BundleContextImpl) l;
            publishFrameworkEvent(
                FrameworkEvent.ERROR, context.bundle, t);
        }
```

```
        }
    }

    private String[] noMatches(boolean optional)
        throws BundleException {
        if (optional) {
            return null;
        }
        throw new BundleException(
            Msg.BUNDLE_NATIVECODE_MATCH_EXCEPTION);
    }
    //设置上下文类加载器。
    private void initializeContextFinder() {
        Thread current = Thread.currentThread();
        Throwable error = null;
        try {
            ClassLoader parent = null;
            String type = FrameworkProperties.getProperty(
                PROP_CONTEXTCLASSLOADER_PARENT);
            if (CONTEXTCLASSLOADER_PARENT_APP.equals(type))
                parent = ClassLoader.getSystemClassLoader();
            else if (CONTEXTCLASSLOADER_PARENT_BOOT.equals(type))
                parent = null;
            else if (CONTEXTCLASSLOADER_PARENT_FWK.equals(type))
                parent = Framework.class.getClassLoader();
            else if (CONTEXTCLASSLOADER_PARENT_EXT.equals(type)) {
                ClassLoader appCL = ClassLoader.getSystemClassLoader();
                if (appCL != null)
                    parent = appCL.getParent();
            } else { //默认是CCL。
```

```
                Method getContextClassLoader = Thread.class.getMethod(
                    "getContextClassLoader", null);
                parent = (ClassLoader) getContextClassLoader.invoke(
                    current, null);
            }
            Method setContextClassLoader = Thread.class.getMethod(
                "setContextClassLoader",
                new Class[] {ClassLoader.class});
            Object[] params = new Object[] {new ContextFinder(parent)};
            setContextClassLoader.invoke(current, params);
            return;
        } catch (SecurityException e) {
            error = e;
        } catch (NoSuchMethodException e) {
            //忽略。
            return;
        } catch (IllegalArgumentException e) {
            error = e;
        } catch (IllegalAccessException e) {
            error = e;
        } catch (InvocationTargetException e) {
            error = e.getTargetException();
        }
    }

    //
    //其它方法，ContextFactoryHandler、URLStreamHandler等。
    //
}
```

3.1.3 AbstractBundle

这个对象用于封装一个内部 Bundle 对象。当一个 Bundle 被卸载后，它被销毁；当被更新后，它可以重用。这个类是抽象类，由 BundleHost 和 BundleFragment 扩展。

```
package org.eclipse.osgi.framework.internal.core;


import java.io.IOException;

import java.io.InputStream;

import java.net.URL;

import java.net.URLConnection;

import java.security.*;

import java.util.*;

import org.eclipse.osgi.framework.adaptor.*;

import org.eclipse.osgi.framework.debug.Debug;

import org.eclipse.osgi.framework.util.KeyedElement;

import org.eclipse.osgi.service.resolver.*;

import org.eclipse.osgi.util.NLS;

import org.osgi.framework.*;


public abstract class AbstractBundle implements Bundle, Comparable,

KeyedElement {

    protected Framework framework;

    protected volatile int state;

    //标识状态正在改变。

    protected volatile Thread stateChanging;

    //Bundle数据对象。

    protected BundleData bundledata;

    //状态更改锁。

    protected Object statechangeLock = new Object();
```

```
protected BundleProtectionDomain domain;

protected ManifestLocalization manifestLocalization = null;

//构建一个Bundle。新建一个BundleHost或BundleFragment。
protected static AbstractBundle createBundle(
    BundleData bundledata, Framework framework)
    throws BundleException {
    if ((bundledata.getType() & BundleData.TYPE_FRAGMENT) > 0)
        return new BundleFragment(bundledata, framework);
    return new BundleHost(bundledata, framework);
}

//构造器。
protected AbstractBundle(BundleData bundledata, Framework framework)
{
    state = INSTALLED;
    stateChanging = null;
    this.bundledata = bundledata;
    this.framework = framework;
    bundledata.setBundle(this);
}

//装载Bundle。
protected abstract void load();
//重新加载新Bundle。必须在拥有Bundle锁时调用。
protected abstract boolean reload(AbstractBundle newBundle);
//刷新Bundle，由Framework.refershPackages调用。
protected abstract void refresh();
//卸载Bundle。
```

```
protected abstract boolean unload();

//关闭Bundle。
protected void close() {
    state = UNINSTALLED;
}
//装载和初始化激活器。
protected BundleActivator loadBundleActivator()
    throws BundleException {
    String activatorClassName = bundledata.getActivator();
    if (activatorClassName != null) {
        try {
            Class activatorClass = loadClass(
                activatorClassName, false);
            return (BundleActivator) (activatorClass.newInstance());
        } catch (Throwable t) { //激活器创建异常。
            throw new BundleException(NLS.bind(
                Msg.BUNDLE_INVALID_ACTIVATOR_EXCEPTION,
                activatorClassName,
                bundledata.getSymbolicName()), t);
        }
    }
    return (null);
}
//装载类。
protected abstract Class loadClass(
    String name, boolean checkPermission)
    throws ClassNotFoundException;


//获取状态。
public int getState() {
```

```
    return (state);
}
//是否激活。
protected boolean isActive() {
    return ((state & (ACTIVE | STARTING)) != 0);
}
//是否解析。
protected boolean isResolved() {
    return (state & (INSTALLED | UNINSTALLED)) == 0;
}
//启动当前Bundle。
//如果当前启动级别小于Bundle启动级别，那么框架必须框架必须持久标记该Bundle
//为started且推迟Bundle启动直到框架当前启动级别大于或等于Bundle启动级别。
//否则，需要完成以下步骤来启动Bundle:
//（1）如果Bundle处于UNINSTALLED状态，则抛出IllegalStateException。
//（2）如果Bundle是ACTIVE或STARTING，那么这个方法直接返回。
//（3）如果这个Bundle是STOPPING，那么这个方法将等待Bundle回到RESOLVED状态
//后再继续。
//（4）如果它发生在一个不合适时机，将抛出一个BundleException。
//（5）如果Bundle不是RESOLVED，则尝试解析Bundle。如果不能解析，则抛出
//BundleException。
//（6）设置状态为STARTTING，调用BundleActivator.start，如果激活器调用失败，
//则状态设回RESOLVED，并调用stop方法，然后抛出BundleException。
//（7）Bundle启动成功后，将记录，如果Framework重启，它将自动重启。
//（8）修改状态为ACTIVE，发布BundleEvent。
//前提条件：getState()为INSTALLED或RESOLVED;
//结果：getState()为ACTIVE，若不为STARTING或ACTIVE，抛出异常。
public void start() throws BundleException {
    start(0);
}
```

```
public void start(int options) throws BundleException {
    framework.checkAdminPermission(this, AdminPermission.EXECUTE);
    checkValid();
    beginStateChange();
    try {
        startWorker(options);
    } finally {
        completeStateChange();
    }
}
//内部启动工作。
protected abstract void startWorker(int options)
    throws BundleException;

//如果Bundle是片段、不是正确启动级别、没有启动持久标记，返回false；
//激活策略为持久忽略、没有定义激活政策返回true。
protected boolean readyToResume() {
    return false;
}
//启动Bundle，而不需要设置持久启动标记。
protected void resume() throws BundleException {
    if (state == UNINSTALLED) {
        return;
    }
    beginStateChange();
    try {
        if (readyToResume())
            startWorker(START_TRANSIENT);
    } finally {
```

```
            completeStateChange();
        }
    }
    //停止Bundle。
    //（1）如果Bundle是UNINSTALLED，抛出IllegalStateException。
    //（2）如果Bundle是STOPPING、RESOLVED或INSTALLED，直接返回。
    //（3）如果Bundle是STARTING，则等待，并继续。
    //（4）如果停止时机不合适，则抛出BundleException。
    //（5）状态设置为STOPPING，记录Bundle已经被停止，
    //因此当框架重启后，它不会被启动。
    //（6）调用BundleActivator.stop，如果异常，则抛出BundleException。
    //（7）卸载监听器、服务和其它使用资源。
    public void stop() throws BundleException {
        stop(0);
    }

    public void stop(int options) throws BundleException {
        framework.checkAdminPermission(this, AdminPermission.EXECUTE);
        checkValid();
        beginStateChange();
        try {
            stopWorker(options);
        } finally {
            completeStateChange();
        }
    }
    //停止Bundle的工作器。
    protected abstract void stopWorker(int options)
        throws BundleException;
    //设置持久状态。State为true，则设置，否则为清理状态位。
```

```
protected void setStatus(final int mask, final boolean state) {
    try {
        AccessController.doPrivileged(
            new PrivilegedExceptionAction() {
            public Object run() throws BundleException, IOException {
                int status = bundledata.getStatus();
                boolean test = ((status & mask) != 0);
                if (test != state) {
                    bundledata.setStatus(
                        state ? (status | mask) : (status & ~mask));
                    bundledata.save();
                }
                return null;
            }
        });
    } catch (PrivilegedActionException pae) {
        framework.publishFrameworkEvent(
            FrameworkEvent.ERROR, this, pae.getException());
    }
}
//停止这个Bundle而不进行标记。
protected void suspend(boolean lock) throws BundleException {
    if (state == UNINSTALLED) {
        return;
    }
    beginStateChange();
    try {
        stopWorker(STOP_TRANSIENT);
    } finally {
        if (!lock) {
```

```
                completeStateChange();
            }
        }
    }
    //更新Bundle。如果Bundle处于激活状态，Bundle将自动停止且更新成功后自动启动。
    //（1）如果Bundle处于UINISTALLED状态，则抛出IllegalStateException。
    //（2）如果Bundle处于ACTIVE或STARTING，Bundle将先stop;
    //如果stop异常，则直接抛出异常。
    //（3）获取新版本，并安装。如果框架不能安装新版本，那么将恢复原Bundle，且在完
    //成以下步骤后抛出异常：状态设置INSTALLED；如果安装成功新版本后，
    //发布BundleEvent；如果原Bundle为ACTIVE，则start它。如果抛出异常，而发布
    //FrameworkEvent事件。
    public void update() throws BundleException {
        framework.checkAdminPermission(
            this, AdminPermission.LIFECYCLE);
            if ((bundledata.getType() &
                (BundleData.TYPE_BOOTCLASSPATH_EXTENSION |
                BundleData.TYPE_FRAMEWORK_EXTENSION)) != 0)
            //权限。
            framework.checkAdminPermission(
                this, AdminPermission.EXTENSIONLIFECYCLE);
        checkValid();
        beginStateChange();
        try {
            final AccessControlContext callerContext =
                AccessController.getContext();

            updateWorker(new PrivilegedExceptionAction() {
                public Object run() throws BundleException {
                    String updateLocation = bundledata.getLocation();
```

```
                if (bundledata.getManifest().get(
                    Constants.BUNDLE_UPDATELOCATION) != null) {
                    updateLocation = (String) bundledata.getManifest(
                        ).get(Constants.BUNDLE_UPDATELOCATION);
                }
                URLConnection source = framework.adaptor.
                    mapLocationToURLConnection(updateLocation);
                //执行更新。
                updateWorkerPrivileged(source, callerContext);
                return null;
            }
        });
    } finally {
        completeStateChange();
    }
}

public void update(final InputStream in) throws BundleException {
    framework.checkAdminPermission(
        this, AdminPermission.LIFECYCLE);
    if ((bundledata.getType() &
         (BundleData.TYPE_BOOTCLASSPATH_EXTENSION |
        BundleData.TYPE_FRAMEWORK_EXTENSION)) != 0)
        framework.checkAdminPermission(this,
            AdminPermission.EXTENSIONLIFECYCLE);
    checkValid();
    beginStateChange();
    try {
        final AccessControlContext callerContext =
            AccessController.getContext();
```

```
            updateWorker(new PrivilegedExceptionAction() {
                public Object run() throws BundleException {
                    URLConnection source = new BundleSource(in);
                    updateWorkerPrivileged(source, callerContext);
                    return null;
                }
            });
        } finally {
            completeStateChange();
        }
    }
    //更新工作器。假定已经获得状态变更锁。
    protected void updateWorker(PrivilegedExceptionAction action)
        throws BundleException {
        //停止Bundle。
        boolean bundleActive = false;
        if (!isFragment())
            bundleActive = (state & (ACTIVE | STARTING)) != 0;
        if (bundleActive) {
            try {
                stopWorker(STOP_TRANSIENT);
            } catch (BundleException e) {
                framework.publishFrameworkEvent(
                    FrameworkEvent.ERROR, this, e);
                if ((state & (ACTIVE | STARTING)) != 0) //如果停止失败。
                {
                    throw e;
                }
            }
        }
```

```
        //执行更新。
        try {
            AccessController.doPrivileged(action);
            framework.publishBundleEvent(BundleEvent.UPDATED, this);
        } catch (PrivilegedActionException pae) {
            if (pae.getException() instanceof RuntimeException)
                throw (RuntimeException) pae.getException();
            throw (BundleException) pae.getException();
        } finally {
            if (bundleActive) {//启动Bundle。
                try {
                    startWorker(START_TRANSIENT);
                } catch (BundleException e) {
                    framework.publishFrameworkEvent(
                        FrameworkEvent.ERROR, this, e);
                }
            }
        }
    }
    //更新。
    protected void updateWorkerPrivileged(URLConnection source,
        AccessControlContext callerContext) throws BundleException {
        AbstractBundle oldBundle = AbstractBundle.createBundle(
            bundledata, framework);
        boolean reloaded = false;
        BundleOperation storage = framework.adaptor.updateBundle(
            this.bundledata, source);
        BundleRepository bundles = framework.getBundles();
        try {
            BundleData newBundleData = storage.begin();
```

```
final AbstractBundle newBundle = framework.
    createAndVerifyBundle(newBundleData);
String[] nativepaths = framework.
    selectNativeCode(newBundle);
if (nativepaths != null) {
    newBundleData.installNativeCode(nativepaths);
}
boolean exporting;
int st = getState();
//装载新Bundle。
synchronized (bundles) {
    exporting = reload(newBundle);
    manifestLocalization = null;
}
reloaded = true;
if (System.getSecurityManager() != null) {
    final boolean extension = (bundledata.getType() &
        (BundleData.TYPE_BOOTCLASSPATH_EXTENSION |
        BundleData.TYPE_FRAMEWORK_EXTENSION)) != 0;
    if (extension && !hasPermission(new AllPermission()))
        throw new BundleException(
            Msg.BUNDLE_EXTENSION_PERMISSION,
            new SecurityException(
                Msg.BUNDLE_EXTENSION_PERMISSION));
    try {
        AccessController.doPrivileged(
            new PrivilegedExceptionAction() {
            public Object run() throws Exception {
                framework.checkAdminPermission(
                    newBundle, AdminPermission.LIFECYCLE);
```

```
                    if (extension)
                        framework.checkAdminPermission(
                        newBundle,
                        AdminPermission.EXTENSIONLIFECYCLE);
                    return null;
                }
            }, callerContext);
        } catch (PrivilegedActionException e) {
            throw e.getException();
        }
    }
    if (st == RESOLVED)
        framework.publishBundleEvent(
            BundleEvent.UNRESOLVED, this);
    storage.commit(exporting);
} catch (Throwable t) {
    try {
        storage.undo();
        if (reloaded)
            synchronized (bundles) {
                reload(oldBundle);
            }
        }
    } catch (BundleException ee) {
        framework.publishFrameworkEvent(
            FrameworkEvent.ERROR, this, ee);
    }
    if (t instanceof SecurityException)
        throw (SecurityException) t;
    if (t instanceof BundleException)
```

```
                throw (BundleException) t;
            throw new BundleException(t.getMessage(), t);
        }
    }
    //卸载Bundle，删除Bundle所有痕迹，包括任何在持久存储区域的数据。
    //（1）如果Bundle在UNINSTALLED状态，抛出IllegalStateException。
    //（2）如果Bundle处于ACTIVE或STARTING，先调用stop方法，若其异常，则广播
    //FrameworkEvent.ERROR。
    //（3）状态设置为UNINSTALLED，删除痕迹。
    public void uninstall() throws BundleException {
        framework.checkAdminPermission(
            this, AdminPermission.LIFECYCLE);
        if ((bundledata.getType() &
            (BundleData.TYPE_BOOTCLASSPATH_EXTENSION |
            BundleData.TYPE_FRAMEWORK_EXTENSION)) != 0)
            framework.checkAdminPermission(this,
                AdminPermission.EXTENSIONLIFECYCLE);
        checkValid();
        beginStateChange();
        try {
            uninstallWorker(new PrivilegedExceptionAction() {
                public Object run() throws BundleException {
                    uninstallWorkerPrivileged();
                    return null;
                }
            });
        } finally {
            completeStateChange();
        }
    }
```

```
//卸载工作器。
protected void uninstallWorker(PrivilegedExceptionAction action)
    throws BundleException {
    boolean bundleActive = false;
    //停止Bundle。
    if (!isFragment())
        bundleActive = (state & (ACTIVE | STARTING)) != 0;
    if (bundleActive) {
        try {
            stopWorker(STOP_TRANSIENT);
        } catch (BundleException e) {
            framework.publishFrameworkEvent(
                FrameworkEvent.ERROR, this, e);
        }
    }
    //执行卸载活动。
    try {
        AccessController.doPrivileged(action);
    } catch (PrivilegedActionException pae) {
        if (bundleActive) { //如果卸载失败，则恢复。
            try {
                startWorker(START_TRANSIENT);
            } catch (BundleException e) {
                //如果启动失败，则麻烦了。
                framework.publishFrameworkEvent(
                    FrameworkEvent.ERROR, this, e);
            }
        }
        throw (BundleException) pae.getException();
    }
```

```
        framework.publishBundleEvent(BundleEvent.UNINSTALLED, this);
    }
    //执行卸载活动。
    protected void uninstallWorkerPrivileged() throws BundleException {
        boolean unloaded = false;
        //缓存头信息。
        getHeaders();
        BundleOperation storage = framework.adaptor.
            uninstallBundle(this.bundledata);
        BundleRepository bundles = framework.getBundles();
        try {
            storage.begin();  //执行卸载。
            boolean exporting;
            int st = getState();
            synchronized (bundles) {
                bundles.remove(this); //删除Bundle。
                exporting = unload();
            }
            if (st == RESOLVED)
                framework.publishBundleEvent(
                    BundleEvent.UNRESOLVED, this);
            unloaded = true;
            storage.commit(exporting); //提交。
            close();
        } catch (BundleException e) {
            try {
                storage.undo();
                if (unloaded) {
                    synchronized (bundles) {
                        load();
```

```
                    bundles.add(this);
                }
            }
        } catch (BundleException ee) {
            //如果装载失败，则麻烦了。
            framework.publishFrameworkEvent(
                FrameworkEvent.ERROR, this, ee);
        }
        throw e;
    } finally {
    }
}
//获取头信息。
public Dictionary getHeaders() {
    return getHeaders(null);
}
public Dictionary getHeaders(String localeString) {
    framework.checkAdminPermission(
        this, AdminPermission.METADATA);
    try {
        initializeManifestLocalization();
    } catch (BundleException e) {
        framework.publishFrameworkEvent(
            FrameworkEvent.ERROR, this, e);
        return new Hashtable();
    }
    if (localeString == null)
        localeString = Locale.getDefault().toString();
    return manifestLocalization.getHeaders(localeString);
}
```

```
//Bundle ID是唯一的，只有UNINSTALLED后，它才销毁。
public long getBundleId() {
    return (bundledata.getBundleID());
}
//获取Location标识。
public String getLocation() {
    framework.checkAdminPermission(
        this, AdminPermission.METADATA);
    return (bundledata.getLocation());
}


public boolean hasPermission(Object permission) {
    checkValid();
    if (domain != null) {
        if (permission instanceof Permission) {
            SecurityManager sm = System.getSecurityManager();
            if (sm instanceof FrameworkSecurityManager) {
                AccessControlContext acc = new AccessControlContext(
                    new ProtectionDomain[] {domain});
                try {
                    sm.checkPermission((Permission) permission, acc);
                    return true;
                } catch (Exception e) {
                    return false;
                }
            }
            return domain.implies((Permission) permission);
        }
        return false;
    }
```

```
    return true;
}
//这个方法标记Bundle的状态为正在更改，因此，start/stop/suspend/update/
//uninstall可以等待状态变更完成。如果调用时stateChanging不为空，则需要等待，
//如果超时，则需要抛出BundleException。调用这个方法的线程，必须用finally
//调用completeStateChange。
protected void beginStateChange() throws BundleException {
    synchronized (statechangeLock) {
        boolean doubleFault = false;
        while (true) {
            if (stateChanging == null) {
                stateChanging = Thread.currentThread();
                return;
            }
            //抛出锁定异常。
            if (doubleFault ||
                (stateChanging == Thread.currentThread())) {
                throw new BundleException(
                    NLS.bind(Msg.BUNDLE_STATE_CHANGE_EXCEPTION,
                    getBundleData().getLocation(),
                    stateChanging.getName()),
                    new BundleStatusException(null,
                    StatusException.CODE_WARNING, stateChanging));
            }
            try {
                long start = 0;
                    start = System.currentTimeMillis();
                }
                statechangeLock.wait(5000); //等等其它线程。
                }
```

```
                } catch (InterruptedException e) {
                }
                doubleFault = true;
            }
        }
    }
    //设置stateChanging为空。
    protected void completeStateChange() {
        synchronized (statechangeLock) {
            if (stateChanging != null) {
                stateChanging = null;
                statechangeLock.notify();
            }
        }
    }
    //Bundle表示。
    public String toString() {
        return (bundledata.getLocation() + " [" + getBundleId() + "]");
    }
    public int compareTo(Object obj) {
        int slcomp = getStartLevel() -
            ((AbstractBundle) obj).getStartLevel();
        if (slcomp != 0) {
            return slcomp;
        }
        long idcomp = getBundleId() - ((AbstractBundle)
            obj).getBundleId();
        return (idcomp < 0L) ? -1 : ((idcomp > 0L) ? 1 : 0);
    }
```

```
//检查状态是否不为UNINSTALLED。
protected void checkValid() {
    if (state == UNINSTALLED) {
        throw new IllegalStateException(
            NLS.bind(Msg.BUNDLE_UNINSTALLED_EXCEPTION,
            getBundleData().getLocation()));
    }
}
//保护域。
protected BundleProtectionDomain getProtectionDomain() {
    return domain;
}
//销毁权限。
protected void unresolvePermissions() {
    if (domain != null) {
        BundlePermissionCollection collection =
            (BundlePermissionCollection) domain.getPermissions();
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            Debug.println("Unresolving permissions in bundle " +
                this);
        }
        collection.unresolvePermissions();
    }
}
//获取片段。
protected Bundle[] getFragments() {
    checkValid();
    return null;
}
//是否片段。
```

```
    protected boolean isFragment() {
        return false;
    }
    //获取宿主。
    protected BundleLoaderProxy[] getHosts() {
        checkValid();
        return null;
    }
    //装载类。
    public Class loadClass(String classname) throws
ClassNotFoundException {
        return loadClass(classname, true);
    }
    //获取EntryPaths。
    public Enumeration getEntryPaths(final String path) {
        try {
            framework.checkAdminPermission(this,
                AdminPermission.RESOURCE);
        } catch (SecurityException e) {
            return null;
        }
        checkValid();
        return (Enumeration) AccessController.doPrivileged(
            new PrivilegedAction() {
            public Object run() {
                return bundledata.getEntryPaths(path);
            }
        });
    }
    //获取Entry。
```

```
public URL getEntry(String fileName) {
    try {
        framework.checkAdminPermission(this,
            AdminPermission.RESOURCE);
    } catch (SecurityException e) {
        return null;
    }
    checkValid();
    if (System.getSecurityManager() == null)
        return bundledata.getEntry(fileName);
    final String ffileName = fileName;
    return (URL) AccessController.doPrivileged(
        new PrivilegedAction() {
        public Object run() {
            return bundledata.getEntry(ffileName);
        }
    });
}

public String getSymbolicName() {
    return bundledata.getSymbolicName();
}

public long getLastModified() {
    return bundledata.getLastModified();
}

public BundleData getBundleData() {
    return bundledata;
}
```

```
public Version getVersion() {
    return bundledata.getVersion();
}

protected BundleDescription getBundleDescription() {
    return framework.adaptor.getState().getBundle(getBundleId());
}

protected int getStartLevel() {
    return bundledata.getStartLevel();
}

protected abstract BundleLoader getBundleLoader();

//标记为RESOLVED。
protected void resolve() {
    if (state == INSTALLED) {
        state = RESOLVED;
        //不发布时间。
    }
}
//获取上下文。
public BundleContext getBundleContext() {
    framework.checkAdminPermission(this, AdminPermission.CONTEXT);
    return getContext();
}
abstract protected BundleContextImpl getContext();

public String getResolutionFailureMessage() {
```

```
        BundleDescription bundleDescription = getBundleDescription();
        if (bundleDescription == null)
            return Msg.BUNDLE_UNRESOLVED_EXCEPTION;
        if (bundleDescription.isResolved())
            throw new IllegalStateException(
                Msg.BUNDLE_UNRESOLVED_STATE_CONFLICT);
        return NLS.bind(
            Msg.BUNDLE_UNRESOLVED_UNSATISFIED_CONSTRAINT_EXCEPTION,
            getResolverError(bundleDescription));
    }

    private String getResolverError(BundleDescription bundleDesc) {
        ResolverError[] errors = framework.adaptor.getState(
            ).getResolverErrors(bundleDesc);
        if (errors == null || errors.length == 0)
            return Msg.BUNDLE_UNRESOLVED_EXCEPTION;
        StringBuffer message = new StringBuffer();
        for (int i = 0; i < errors.length; i++) {
            message.append(errors[i].toString());
            if (i < errors.length - 1)
                message.append(", ");
        }
        return message.toString();
    }

    public int getKeyHashCode() {
        return (int) getBundleId();
    }

    public boolean compare(KeyedElement other) {
```

```
    return getBundleId() == ((AbstractBundle) other).getBundleId();
}

public Object getKey() {
    return new Long(getBundleId());
}
//获取本地化RB。
public ResourceBundle getResourceBundle(String localeString) {
    try {
        initializeManifestLocalization();
    } catch (BundleException ex) {
        return (null);
    }
    if (localeString == null) {
        localeString = Locale.getDefault().toString();
    }
    return manifestLocalization.getResourceBundle(localeString);
}

private void initializeManifestLocalization()
throws BundleException {
    if (manifestLocalization == null) {
        Dictionary rawHeaders;
        rawHeaders = bundledata.getManifest();
        manifestLocalization = new ManifestLocalization(
            this, rawHeaders);
    }
}

public boolean testStateChanging(Object thread) {
```

```
    return stateChanging == thread;
}

public Thread getStateChanging() {
    return stateChanging;
}
//查找资源。
public Enumeration findEntries(String path, String filePattern,
    boolean recurse) {
    try {
        framework.checkAdminPermission(this,
            AdminPermission.RESOURCE);
    } catch (SecurityException e) {
        return null;
    }
    checkValid();
    if (!isResolved())
        framework.packageAdmin.resolveBundles(
            new Bundle[] {this});
    List pathList = new ArrayList();
    Filter patternFilter = null;
    Hashtable patternProps = null;
    if (filePattern != null)
        try {
            patternFilter = new FilterImpl("(filename=" +
                filePattern + ")");
            patternProps = new Hashtable(2);
        } catch (InvalidSyntaxException e) {
        }
    //查找本地。
```

```
findLocalEntryPaths(path, patternFilter, patternProps, recurse,
    pathList);
//查找片段。
final Bundle[] fragments = getFragments();
final int numFragments = fragments == null ? -1 : fragments.length;
for (int i = 0; i < numFragments; i++)
    ((AbstractBundle) fragments[i]).findLocalEntryPaths(
        path, patternFilter, patternProps, recurse, pathList);
if (pathList.size() == 0)
    return null;
final String[] pathArray = (String[]) pathList.toArray(
    new String[pathList.size()]);
return new Enumeration() {
    int curIndex = 0;
    int curFragment = -1;
    URL nextElement = null;

    public boolean hasMoreElements() {
        if (nextElement != null)
            return true;
        getNextElement();
        return nextElement != null;
    }

    public Object nextElement() {
        if (!hasMoreElements())
            throw new NoSuchElementException();
        URL result;
        result = nextElement;
        getNextElement();
```

```
            return result;
        }

        private void getNextElement() {
            nextElement = null;
            if (curIndex >= pathArray.length)
                return;
            String curPath = pathArray[curIndex];
            if (curFragment == -1) {
                nextElement = getEntry(curPath);
                curFragment++;
            }
            while (nextElement == null && curFragment < numFragments)
                nextElement =
                    fragments[curFragment++].getEntry(curPath);
            if (numFragments == -1 || curFragment >= numFragments) {
                curIndex++;
                curFragment = -1;
            }
            if (nextElement == null)
                getNextElement();
        }
    };
}

protected void findLocalEntryPaths(String path,
    Filter patternFilter, Hashtable patternProps,
    boolean recurse, List pathList) {
    Enumeration entryPaths = bundledata.getEntryPaths(path);
    if (entryPaths == null)
```

```
            return;
        while (entryPaths.hasMoreElements()) {
            String entry = (String) entryPaths.nextElement();
            int lastSlash = entry.lastIndexOf('/');
            if (patternProps != null) {
                int secondToLastSlash = entry.lastIndexOf('/',
                    lastSlash - 1);
                int fileStart;
                int fileEnd = entry.length();
                if (lastSlash < 0)
                    fileStart = 0;
                else if (lastSlash != entry.length() - 1)
                    fileStart = lastSlash + 1;
                else {
                    fileEnd = lastSlash;
                    if (secondToLastSlash < 0)
                        fileStart = 0;
                    else
                        fileStart = secondToLastSlash + 1;
                }
                String fileName = entry.substring(fileStart, fileEnd);
                patternProps.put("filename", fileName);
            }
            if (!pathList.contains(entry) && (patternFilter == null ||
                patternFilter.matchCase(patternProps)))
                pathList.add(entry);
            if (recurse && !entry.equals(path) && entry.length() > 0 &&
                lastSlash == (entry.length() - 1))
                findLocalEntryPaths(entry, patternFilter, patternProps,
                    recurse, pathList);
```

```
        }
        return;
    }

    class BundleStatusException extends Throwable
    implements StatusException {
        private int code;
        private Object status;
        BundleStatusException(String message, int code, Object status) {
            super(message);
            this.code = code;
            this.status = status;
        }
        public Object getStatus() {
            return status;
        }
        public int getStatusCode() {
            return code;
        }

    }
}
```

3.1.4 BundleHost

```
package org.eclipse.osgi.framework.internal.core;

import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;
```

```
import org.eclipse.osgi.framework.adaptor.*;
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.framework.log.FrameworkLogEntry;
import org.eclipse.osgi.service.resolver.BundleDescription;
import org.eclipse.osgi.util.NLS;
import org.osgi.framework.*;

public class BundleHost extends AbstractBundle {
    //BundleLoader的代理，允许BundleLoader的晚创建。
    private BundleLoaderProxy proxy;
    //上下文。
    protected BundleContextImpl context;
    //片段。
    protected BundleFragment[] fragments;
    //新建一个BundleHost。
    public BundleHost(BundleData bundledata, Framework framework)
        throws BundleException {
        super(bundledata, framework);
        context = null;
        fragments = null;
    }
    //装载Bundle。
    protected void load() {
        //框架为激活下装载。
        if (framework.isActive()) {
            SecurityManager sm = System.getSecurityManager();
            if (sm != null && framework.permissionAdmin != null) {
                domain = framework.permissionAdmin.
                    createProtectionDomain(this);
            }
```

```
        }
        proxy = null;
    }
    //重新加载。
    protected boolean reload(AbstractBundle newBundle) {
        boolean exporting = false;

        if (framework.isActive()) {
            if (state == RESOLVED) {
                BundleLoaderProxy curProxy = getLoaderProxy();
                exporting = curProxy.inUse();
                //如果正在使用，确保BundleLoader已经创建。
                if (exporting)
                    curProxy.getBundleLoader().createClassLoader();
                else
                    closeBundleLoader(proxy);
                state = INSTALLED;
                proxy = null;
                fragments = null;
            }

        } else {
            //关闭。
            try {
                this.bundledata.close();
            } catch (IOException e) {
            }
        }
        this.bundledata = newBundle.bundledata;
        this.bundledata.setBundle(this);
```

```
    if (framework.isActive() && System.getSecurityManager() != null
            && framework.permissionAdmin != null)
        domain = framework.permissionAdmin.
            createProtectionDomain(this);
    return (exporting);
}
//刷新Bundle，由Framework.refreshPackages调用。
protected void refresh() {
    if (state == RESOLVED) {
        closeBundleLoader(proxy);
        proxy = null;
        fragments = null;
        state = INSTALLED;
    }
    manifestLocalization = null;
}
//Unload Bundle。关闭BundleData。
protected boolean unload() {
    boolean exporting = false;

    if (framework.isActive()) {
        if (state == RESOLVED) {
            BundleLoaderProxy curProxy = getLoaderProxy();
            exporting = curProxy.inUse();
            if (exporting)
                curProxy.getBundleLoader().createClassLoader();
            else
                closeBundleLoader(proxy);

            state = INSTALLED;
```

```
                proxy = null;
                fragments = null;
                domain = null;
            }
        }
        if (!exporting) {
            try {
                this.bundledata.close();
            } catch (IOException e) {
            }
        }

        return (exporting);
    }
    //检查是否解析，并返回Loader。
    private BundleLoader checkLoader() {
        checkValid();
        //检查Bundle是否已经解析。
        if (!isResolved()) {
            if (!framework.packageAdmin.resolveBundles(
                    new Bundle[] {this})) {
                return null;  //解析失败。
            }
        }

        BundleLoader loader = getBundleLoader();
        if (loader == null) {
            return null;
        }
        return loader;
```

```
}
//装载类。
protected Class loadClass(String name, boolean checkPermission)
    throws ClassNotFoundException {
    if (checkPermission) {
        try {
            framework.checkAdminPermission(
                this, AdminPermission.CLASS);
        } catch (SecurityException e) {
            throw new ClassNotFoundException();
        }
    }
    BundleLoader loader = checkLoader();
    if (loader == null)
        throw new ClassNotFoundException(
            NLS.bind(Msg.BUNDLE_CNFE_NOT_RESOLVED, name,
            getBundleData().getLocation()));
    try {
        //由Loader加载类。
        return (loader.loadClass(name));
    } catch (ClassNotFoundException e) {
        if (!(e instanceof StatusException) &&
            (bundledata.getStatus() &
            Constants.BUNDLE_LAZY_START) != 0
            && !testStateChanging(Thread.currentThread()))
            try {
                framework.secureAction.start(this, START_TRANSIENT);
            } catch (BundleException be) {
                framework.adaptor.getFrameworkLog().log(
                    new FrameworkLogEntry(
```

```
                        FrameworkAdaptor.FRAMEWORK_SYMBOLICNAME,
                        FrameworkLogEntry.WARNING, 0,
                        be.getMessage(), 0, be, null));
            }
            throw e;
        }
    }
    //获取资源。首先，检查权限；其次，调用Loader获取资源。
    public URL getResource(String name) {
        BundleLoader loader = null;
        try {
            framework.checkAdminPermission(
                this, AdminPermission.RESOURCE);
        } catch (SecurityException ee) {
            return null;
        }
        loader = checkLoader();
        if (loader == null)
            return null;
        return (loader.findResource(name));
    }
    //同上。
    public Enumeration getResources(String name) throws IOException {
        BundleLoader loader = null;
        try {
            framework.checkAdminPermission(
                this, AdminPermission.RESOURCE);
        } catch (SecurityException ee) {
            return null;
        }
```

```
        loader = checkLoader();
        if (loader == null)
            return null;
        Enumeration result = loader.getResources(name);
        if (result != null && result.hasMoreElements())
            return result;
        return null;
    }
    //启动Bundle。
    protected void startWorker(int options) throws BundleException {
        //首先，根据Options设置状态。
        if ((options & START_TRANSIENT) == 0) {
            setStatus(Constants.BUNDLE_STARTED, true);
            setStatus(Constants.BUNDLE_ACTIVATION_POLICY,
                (options & START_ACTIVATION_POLICY) != 0);
            if (Debug.DEBUG && Debug.MONITOR_ACTIVATION)
                new Exception("A persistent start has been called on
                    bundle: " + getBundleData()).printStackTrace();
        }
        //如果框架没有激活或Bundle已经激活，则返回。
        if (!framework.active || (state & ACTIVE) != 0)
            return;
        //如果Bundle未解析，则开始解析。
        if (state == INSTALLED) {
            if (!framework.packageAdmin.resolveBundles(
                new Bundle[] {this}))
                throw new BundleException(
                    getResolutionFailureMessage());
        }
        //检查启动级别
```

```
if (getStartLevel() > framework.startLevelManager.
    getStartLevel()){
    if ((options & START_TRANSIENT) != 0) {
        String msg = NLS.bind(
            Msg.BUNDLE_TRANSIENT_START_ERROR, this);
        //晚启动将导致警告。
        throw new BundleException(msg, new BundleStatusException(
            msg, StatusException.CODE_WARNING, this));
    }
    return;
}
//如果使用晚激活政策。
if ((options & START_ACTIVATION_POLICY) != 0 &&
    (state & STARTING) == 0) {
    //必须使用晚激活政策。
    if ((bundledata.getStatus() &
        Constants.BUNDLE_LAZY_START) != 0) {
        //发布事件并返回。
        state = STARTING;
        framework.publishBundleEvent(
            BundleEvent.LAZY_ACTIVATION, this);
        return;
    }
}
//发布正在启动事件。
state = STARTING;
framework.publishBundleEvent(BundleEvent.STARTING, this);
context = getContext();
//开始启动。
long start = 0;
```

```
try {
    //启动。
    context.start();

    if (framework.active) {
        state = ACTIVE;
        //发布已经启动。
        framework.publishBundleEvent(
            BundleEvent.STARTED, this);
    }

} catch (BundleException e) {
    //发布正在停止事件。
    state = STOPPING;
    framework.publishBundleEvent(BundleEvent.STOPPING, this);
    //销毁上下文。
    context.close();
    context = null;
    //修改状态。
    state = RESOLVED;
    //发布事件。
    framework.publishBundleEvent(BundleEvent.STOPPED, this);
    throw e;
} finally {

}
//如果装载变为卸载，则销毁上下文。
if (state == UNINSTALLED) {
    context.close();
    context = null;
```

```
            throw new BundleException(
                NLS.bind(Msg.BUNDLE_UNINSTALLED_EXCEPTION,
                getBundleData().getLocation()));
        }
    }
    //是否可以继续。
    protected boolean readyToResume() {
        //启动级别错误。
        if (getStartLevel() > framework.
            startLevelManager.getStartLevel())
            return false;
        int status = bundledata.getStatus();
        //如果Bundle没有持久标记为启动，则返回false。
        // Return false if the bundle is not persistently marked for start
        if ((status & Constants.BUNDLE_STARTED) == 0)
            return false;
        if ((status & Constants.BUNDLE_ACTIVATION_POLICY) == 0
            || (status & Constants.BUNDLE_LAZY_START) == 0)
            return true;
        //如果未解析。
        if (!isResolved())
            //不能从UNRESOLVED变到STARTING。
            return false;
        //发布晚激活事件。
        state = STARTING;
        framework.publishBundleEvent(
            BundleEvent.LAZY_ACTIVATION, this);
        return false;
    }
```

```
//创建上下文。
protected BundleContextImpl createContext() {
    return (new BundleContextImpl(this));
}
//获取上下文。
protected synchronized BundleContextImpl getContext() {
    if (context == null) {
        //只在STARTING、ACTIVE和STOPPING时创建上下文。
        if ((state & (STARTING | ACTIVE | STOPPING)) != 0)
            context = createContext();
    }
    return (context);
}
//停止一个Bundle。
protected void stopWorker(int options) throws BundleException {
    //设置状态。
    if ((options & STOP_TRANSIENT) == 0) {
        setStatus(Constants.BUNDLE_STARTED, false);
        setStatus(Constants.BUNDLE_ACTIVATION_POLICY, false);
        if (Debug.DEBUG && Debug.MONITOR_ACTIVATION)
            new Exception("A persistent start has been called
            on bundle: " + getBundleData()).printStackTrace();
    }
    if (framework.active) {
        //直接返回。
        if ((state & (STOPPING | RESOLVED | INSTALLED)) != 0) {
            return;
        }

        state = STOPPING;
```

```
            framework.publishBundleEvent(BundleEvent.STOPPING, this);
            try {
                //玩启动的Bundle的状态为STARTING时，上下文为空。
                if (context != null)
                    context.stop();
            } finally {
                if (context != null) {
                    context.close();
                    context = null;
                }

                checkValid();
                //更改状态。
                state = RESOLVED;

                if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
                    Debug.println("->stopped " + this);
                }
                //发布事件。
                framework.publishBundleEvent(
                    BundleEvent.STOPPED, this);
            }
        }
    }
    //获取注册的服务的引用。
    public org.osgi.framework.ServiceReference[]
        getRegisteredServices() {
        checkValid();

        if (context == null) {
```

```
            return (null);
        }

        return (context.getRegisteredServices());
    }
    //正在使用的服务。
    public org.osgi.framework.ServiceReference[] getServicesInUse() {
        checkValid();

        if (context == null) {
            return (null);
        }

        return (context.getServicesInUse());
    }
    //获取片段的拷贝。
    protected Bundle[] getFragments() {
        synchronized (framework.bundles) {
            if (fragments == null)
                return null;
            Bundle[] result = new Bundle[fragments.length];
            System.arraycopy(fragments, 0, result, 0, result.length);
            return result;
        }
    }
    //按ID顺序附加一个片段。
    protected void attachFragment(BundleFragment fragment)
        throws BundleException {
        //获取Loader，但不强迫创建一个。
        BundleLoader loader = getLoaderProxy().getBasicBundleLoader();
```

```
//如果Loader已经创建，则使用Loader。
if (loader != null)
    loader.attachFragment(fragment);


if (fragments == null) {
    fragments = new BundleFragment[] {fragment};
} else {
    boolean inserted = false;


    BundleFragment[] newFragments =
        new BundleFragment[fragments.length + 1];
    for (int i = 0; i < fragments.length; i++) {
        //已经存在，则返回。
        if (fragment == fragments[i])
            return;
        //获取插入的顺序。
        if (!inserted && fragment.getBundleId() <
            fragments[i].getBundleId()) {
            //如果Loader已经创建，则不能附加到链的中间。
            if (loader != null) {
                throw new BundleException(NLS.bind(
                    Msg.BUNDLE_LOADER_ATTACHMENT_ERROR,
                    fragments[i].getSymbolicName(),
                    getSymbolicName()));
            }
            newFragments[i] = fragment;
            inserted = true;
        }
        newFragments[inserted ? i + 1 : i] = fragments[i];
    }
```

```
            if (!inserted)
                newFragments[newFragments.length - 1] = fragment;
            fragments = newFragments;
        }
    }
    //获取Loader。
    protected BundleLoader getBundleLoader() {
        BundleLoaderProxy curProxy = getLoaderProxy();
        return curProxy == null ? null : curProxy.getBundleLoader();
    }
    //获取Loader代理。
    protected synchronized BundleLoaderProxy getLoaderProxy() {
        if (proxy != null)
            return proxy;
        BundleDescription bundleDescription = getBundleDescription();
        if (bundleDescription == null)
            return null;
        proxy = new BundleLoaderProxy(this, bundleDescription);
        bundleDescription.setUserObject(proxy);
        return proxy;
    }
    //关闭Loader。
    static void closeBundleLoader(BundleLoaderProxy proxy) {
        if (proxy == null)
            return;
        //关闭Bundle Loader。
        BundleLoader loader = proxy.getBasicBundleLoader();
        if (loader != null)
            loader.close();
        proxy.setStale();
```

```
        //设置BundleDescription。
        BundleDescription description = proxy.getBundleDescription();
        description.setUserObject(null);
    }
}
```

3.1.5 BundleFragment

```
package org.eclipse.osgi.framework.internal.core;

import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;
import org.eclipse.osgi.framework.adaptor.BundleData;
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.util.NLS;
import org.osgi.framework.*;

public class BundleFragment extends AbstractBundle {
    //Hosts。
    protected BundleLoaderProxy[] hosts;

    public BundleFragment(BundleData bundledata, Framework framework)
        throws BundleException {
        super(bundledata, framework);
        hosts = null;
    }

    //装载。
    protected void load() {
```

```
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            if ((state & (INSTALLED)) == 0) {
                Debug.println("Bundle.load called when state != INSTALLED:
                    " + this);
                Debug.printStackTrace(new Exception("Stack trace"));
            }
        }

        if (framework.isActive()) {
            SecurityManager sm = System.getSecurityManager();

            if (sm != null && framework.permissionAdmin != null) {
                domain = framework.permissionAdmin.
                    createProtectionDomain(this);
            }
        }
    }

    protected boolean reload(AbstractBundle newBundle) {
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            if ((state & (INSTALLED | RESOLVED)) == 0) {
                Debug.println("Bundle.reload called when state !=
                    INSTALLED | RESOLVED: " + this);
                Debug.printStackTrace(new Exception("Stack trace"));
            }
        }

        boolean exporting = false;
        if (framework.isActive()) {
            if (hosts != null) {
```

```
            if (state == RESOLVED) {
                exporting = true;
                hosts = null;
                state = INSTALLED;
            }
        }
    } else {
        try {
            this.bundledata.close();
        } catch (IOException e) {
        }
    }
    if (!exporting) {
        try {
            this.bundledata.close();
        } catch (IOException e) {
        }
    }

    this.bundledata = newBundle.bundledata;
    this.bundledata.setBundle(this);
    if (framework.isActive() && System.getSecurityManager() != null
        && framework.permissionAdmin != null)
        domain = framework.permissionAdmin.
            createProtectionDomain(this);
    return (exporting);
}

//刷新。
protected void refresh() {
```

```
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            if ((state & (UNINSTALLED | INSTALLED | RESOLVED)) == 0) {
                Debug.println("Bundle.refresh called when state !=
                    UNINSTALLED | INSTALLED | RESOLVED: " + this);
                Debug.printStackTrace(new Exception("Stack trace"));
            }
        }

        if (state == RESOLVED) {
            hosts = null;
            state = INSTALLED;
        }
        manifestLocalization = null;
    }

    protected boolean unload() {
        if (Debug.DEBUG && Debug.DEBUG_GENERAL) {
            if ((state & (UNINSTALLED | INSTALLED | RESOLVED)) == 0) {
                Debug.println("Bundle.unload called when state !=
                    UNINSTALLED | INSTALLED | RESOLVED: " + this);
                Debug.printStackTrace(new Exception("Stack trace"));
            }
        }

        boolean exporting = false;
        if (framework.isActive()) {
            if (hosts != null) {
                if (state == RESOLVED) {
                    exporting = true;
                    hosts = null;
```

```
                state = INSTALLED;
            }
            domain = null;
        }
    }
    if (!exporting) {
        try {
            this.bundledata.close();
        } catch (IOException e) {
        }
    }

    return (exporting);
}
//以下操作对片段无效。
protected Class loadClass(String name, boolean checkPermission)
    throws ClassNotFoundException {
    if (checkPermission) {
        try {
            framework.checkAdminPermission(
                this, AdminPermission.CLASS);
        } catch (SecurityException e) {
            throw new ClassNotFoundException();
        }
        checkValid();
    }
    throw new ClassNotFoundException(
        NLS.bind(Msg.BUNDLE_FRAGMENT_CNFE, name));
}
public URL getResource(String name) {
```

```
    checkValid();
    return (null);
}
public Enumeration getResources(String name) {
    checkValid();
    return null;
}
protected void startWorker(int options) throws BundleException {
    throw new BundleException(
        NLS.bind(Msg.BUNDLE_FRAGMENT_START, this));
}
protected void stopWorker(int options) throws BundleException {
    throw new BundleException(
        NLS.bind(Msg.BUNDLE_FRAGMENT_STOP, this));
}
public ServiceReference[] getRegisteredServices() {
    checkValid();
    return null;
}
public ServiceReference[] getServicesInUse() {
    checkValid();
    return null;
}
//获取宿主。
protected BundleLoaderProxy[] getHosts() {
    return hosts;
}
//是否片段返回true。
protected boolean isFragment() {
    return true;
```

```
}

//添加Host。
protected boolean addHost(BundleLoaderProxy host) {
    if (host == null)
        return false;
    try {
        ((BundleHost) host.getBundleHost()).attachFragment(this);
    } catch (BundleException be) {
        framework.publishFrameworkEvent(
            FrameworkEvent.ERROR, host.getBundleHost(), be);
        return false;
    }
    if (hosts == null) {
        hosts = new BundleLoaderProxy[] {host};
        return true;
    }
    for (int i = 0; i < hosts.length; i++) {
        if (host.getBundleHost() == hosts[i].getBundleHost())
            return true; // 已经存在。
    }
    //添加新的Host。
    BundleLoaderProxy[] newHosts = new
        BundleLoaderProxy[hosts.length + 1];
    System.arraycopy(hosts, 0, newHosts, 0, hosts.length);
    newHosts[newHosts.length - 1] = host;
    hosts = newHosts;
    return true;
}
```

```
    protected BundleLoader getBundleLoader() {
        return null;
    }
    protected BundleContextImpl getContext() {
        return null;
    }
}
```

3.1.6 BundleContextImpl

```
package org.eclipse.osgi.framework.internal.core;

import java.io.File;
import java.io.InputStream;
import java.security.*;
import java.util.*;
import org.eclipse.osgi.event.BatchBundleListener;
import org.eclipse.osgi.framework.debug.Debug;
import
org.eclipse.osgi.framework.eventmgr.EventDispatcher;
import org.eclipse.osgi.framework.eventmgr.EventListeners;
import org.eclipse.osgi.internal.profile.Profile;
import org.eclipse.osgi.util.NLS;
import org.osgi.framework.*;

//Bundle执行上下文。当Bundle停止时销毁。
public class BundleContextImpl
    implements BundleContext, EventDispatcher {
    public static final String PROP_SCOPE_SERVICE_EVENTS
```

```
    = "osgi.scopeServiceEvents";
public static final boolean scopeEvents
    = Boolean.valueOf(FrameworkProperties.getProperty(
    PROP_SCOPE_SERVICE_EVENTS, "true")).booleanValue();
//是否可用标识。
private boolean valid;
//框架使用字段直接访问因为当上下文不可用时，它还需要访问。
protected BundleHost bundle;
protected Framework framework;
//Bundle使用的服务。主键是ServiceReference，值是ServiceUse。
protected Hashtable servicesInUse;
//Bundle事件监听器。
protected EventListeners bundleEvent;
protected EventListeners bundleEventSync;
protected EventListeners serviceEvent;
protected EventListeners frameworkEvent;
//激活器。
protected BundleActivator activator;
//锁对象。
protected Object contextLock = new Object();
//为Bundle包装一个BundleContext。
protected BundleContextImpl(BundleHost bundle) {
    this.bundle = bundle;
    valid = true;
    framework = bundle.framework;
    bundleEvent = null;
    bundleEventSync = null;
```

```
        serviceEvent = null;
        frameworkEvent = null;
        servicesInUse = null;
        activator = null;
    }
    //销毁，当Bundle停止时调用。
    protected void close() {
        valid = false; //标识无效。
        //移除所有监听器。
        synchronized (framework.serviceEvent) {
            if (serviceEvent != null) {
                framework.serviceEvent.removeListener(this);
                serviceEvent = null;
            }
        }
        synchronized (framework.frameworkEvent) {
            if (frameworkEvent != null) {
                framework.frameworkEvent.removeListener(this);
                frameworkEvent = null;
            }
        }
        synchronized (framework.bundleEvent) {
            if (bundleEvent != null) {
                framework.bundleEvent.removeListener(this);
                bundleEvent = null;
            }
        }
        synchronized (framework.bundleEventSync) {
            if (bundleEventSync != null) {
```

```
    framework.bundleEventSync.removeListener(this);
            bundleEventSync = null;
        }
      }
      //卸载由当前Bundle注册的服务。
      ServiceReference[] publishedReferences = null;
      synchronized (framework.serviceRegistry) {
         publishedReferences = framework.serviceRegistry.
            lookupServiceReferences(this);
      }

      if (publishedReferences != null) {
         for (int i = 0; i < publishedReferences.length; i++)
{
            try {
               ((ServiceReferenceImpl)
publishedReferences[i]).
                  registration.unregister();
            } catch (IllegalStateException e) {
            }
         }
      }
      //如果该Bundle使用了服务，则需要释放。
      if (servicesInUse != null) {
         int usedSize;
         ServiceReference[] usedRefs = null;
         synchronized (servicesInUse) {
            usedSize = servicesInUse.size();
```

```
            if (usedSize > 0) {
                if (Debug.DEBUG && Debug.DEBUG_SERVICES) {
                    Debug.println("Releasing services");
                }

                usedRefs = new ServiceReference[usedSize];

                Enumeration refsEnum = servicesInUse.keys();
                for (int i = 0; i < usedSize; i++) {
                    usedRefs[i] = (ServiceReference)
                        refsEnum.nextElement();
                }
            }
        }
        //释放服务。
        for (int i = 0; i < usedSize; i++) {
            ((ServiceReferenceImpl)
usedRefs[i]).registration.
                releaseService(this);
        }

        servicesInUse = null;
    }

    bundle = null;
  }
  //获取系统属性。
  public String getProperty(String key) {
```

```
        SecurityManager sm = System.getSecurityManager();

        if (sm != null) {
            sm.checkPropertyAccess(key);
        }

        return (framework.getProperty(key));
    }
    //获取Bundle。
    public org.osgi.framework.Bundle getBundle() {
        checkValid();

        return (bundle);
    }
    //安装Bundle。
    public org.osgi.framework.Bundle installBundle(String
location)
         throws BundleException {
        checkValid();
        //使用Framework安装，它将创建一个Bundle，并调用Install方法。
        return framework.installBundle(location);
    }
    public org.osgi.framework.Bundle installBundle(
        String location, InputStream in) throws
    BundleException {
        checkValid();
        return framework.installBundle(location, in);
    }
```

```
    //获取Bundle。
    public org.osgi.framework.Bundle getBundle(long id) {
        return (framework.getBundle(id));
    }
    public     AbstractBundle     getBundleByLocation(String
location) {
        return (framework.getBundleByLocation(location));
    }
    public org.osgi.framework.Bundle[] getBundles() {
        return framework.getAllBundles();
    }

    public void addServiceListener(ServiceListener listener,
        String filter) throws InvalidSyntaxException {
        checkValid();

        ServiceListener filteredListener =
            new   FilteredServiceListener(filter,   listener,
        this);

        synchronized (framework.serviceEvent) {
            if (serviceEvent == null) {
                serviceEvent = new EventListeners();
                framework.serviceEvent.addListener(this,
this);
            }

            serviceEvent.addListener(listener,
filteredListener);
```

```
        }
    }
    public void addServiceListener(ServiceListener listener)
{
        try {
            addServiceListener(listener, null);
        } catch (InvalidSyntaxException e) {
        }
    }
    public    void    removeServiceListener(ServiceListener
listener) {
        checkValid();

        synchronized (framework.serviceEvent) {
            if (serviceEvent != null) {
                serviceEvent.removeListener(listener);
            }
        }
    }

    public void addBundleListener(BundleListener listener) {
        checkValid();

        if (listener instanceof SynchronousBundleListener) {
            framework.checkAdminPermission(getBundle(),

                AdminPermission.LISTENER);
            synchronized (framework.bundleEventSync) {
                if (bundleEventSync == null) {
                    bundleEventSync = new EventListeners();
```

```
                framework.bundleEventSync.addListener(this,
this);
            }

            bundleEventSync.addListener(listener,
listener);
         }
      } else {
         synchronized (framework.bundleEvent) {
            if (bundleEvent == null) {
               bundleEvent = new EventListeners();
               framework.bundleEvent.addListener(this,
this);
            }

            bundleEvent.addListener(listener, listener);
         }
      }
   }
  public void removeBundleListener(BundleListener listener)
{
      checkValid();

      if (listener instanceof SynchronousBundleListener) {
         framework.checkAdminPermission(getBundle(),
            AdminPermission.LISTENER);

         synchronized (framework.bundleEventSync) {
            if (bundleEventSync != null) {
               bundleEventSync.removeListener(listener);
```

```
            }
         }
      } else {
         synchronized (framework.bundleEvent) {
            if (bundleEvent != null) {
               bundleEvent.removeListener(listener);
            }
         }
      }
   }

   public void addFrameworkListener(FrameworkListener
listener) {
      checkValid();
      synchronized (framework.frameworkEvent) {
         if (frameworkEvent == null) {
            frameworkEvent = new EventListeners();
            framework.frameworkEvent.addListener(this,
this);
         }

         frameworkEvent.addListener(listener, listener);
      }
   }

   public void removeFrameworkListener(FrameworkListener
listener) {
      checkValid();

      synchronized (framework.frameworkEvent) {
```

```
            if (frameworkEvent != null) {
                frameworkEvent.removeListener(listener);
            }
        }
    }
    public org.osgi.framework.ServiceRegistration
registerService(
        String[] clazzes, Object service, Dictionary
    properties) {
        checkValid();

        if (service == null) {
            throw new IllegalArgumentException(
                Msg.SERVICE_ARGUMENT_NULL_EXCEPTION);
        }
        int size = clazzes.length;
        if (size == 0) {
            throw new IllegalArgumentException(
                Msg.SERVICE_EMPTY_CLASS_LIST_EXCEPTION);
        }
        //拷贝以便操作。
        String[] copy = new String[clazzes.length];
        for (int i = clazzes.length - 1; i >= 0; i--)
            copy[i] = clazzes[i].intern();
        clazzes = copy;
        //权限检查。
        framework.checkRegisterServicePermission(clazzes);
        //检查服务是否实现接口。
        if (!(service instanceof ServiceFactory)) {
```

```
            String invalidService = checkServiceClass(
                clazzes, service);
            if (invalidService != null) {
                throw new IllegalArgumentException(
                    NLS.bind(
                        Msg.SERVICE_NOT_INSTANCEOF_CLASS_EXCEPTI
                    ON,
                        invalidService));
            }
        }
        return (createServiceRegistration(
            clazzes, service, properties));
    }


    static String checkServiceClass(
        final String[] clazzes, final Object serviceObject) {
        //获取ServiceObject的CL。
        ClassLoader cl =
            (ClassLoader) AccessController.doPrivileged(
            new PrivilegedAction() {
            public Object run() {
                return
serviceObject.getClass().getClassLoader();
            }
        });
        for (int i = 0; i < clazzes.length; i++) {
            try {
                Class serviceClazz =
```

```
                cl == null ? Class.forName(clazzes[i]) :
                cl.loadClass(clazzes[i]);
            if (!serviceClazz.isInstance(serviceObject))
                return clazzes[i];
        } catch (ClassNotFoundException e) {
            //不太常见。
            if (extensiveCheckServiceClass(clazzes[i],
                serviceObject.getClass()))
                return clazzes[i];
        }
    }
    return null;
}
//显示判断继承关系。
private static boolean extensiveCheckServiceClass(
    String clazz, Class serviceClazz) {
    if (clazz.equals(serviceClazz.getName()))
        return false;
    Class[] interfaces = serviceClazz.getInterfaces();
    for (int i = 0; i < interfaces.length; i++)
        if (!extensiveCheckServiceClass(clazz,
interfaces[i]))
            return false;
    Class superClazz = serviceClazz.getSuperclass();
    if (superClazz != null)
        if (!extensiveCheckServiceClass(clazz,
superClazz))
            return false;
    return true;
```

```
    }
    //创建服务注册信息。
    protected                          ServiceRegistrationImpl
createServiceRegistration(
        String[]   clazzes,   Object   service,   Dictionary
    properties) {
        return (new ServiceRegistrationImpl(
            this, clazzes, service, properties));
    }

    public          org.osgi.framework.ServiceRegistration
registerService(
        String clazz, Object service, Dictionary properties)
    {
        String[] clazzes = new String[] {clazz};
        return       (registerService(clazzes,       service,
properties));
    }
    //获取服务引用。
    public org.osgi.framework.ServiceReference[]
        getServiceReferences(String clazz, String filter)
        throws InvalidSyntaxException {
        checkValid();

        return (framework.getServiceReferences(
            clazz, filter, this, false));
    }

    public                                 ServiceReference[]
```

```
getAllServiceReferences(String clazz,
      String filter) throws InvalidSyntaxException {
      checkValid();
      return (framework.getServiceReferences(
         clazz, filter, this, true));
   }

   public org.osgi.framework.ServiceReference
      getServiceReference(String clazz) {
      checkValid();
      try {
         ServiceReference[] references = framework.
            getServiceReferences(clazz, null, this, false);

         if (references != null) {
            int index = 0;

            int length = references.length;
            //选择优先级更高的。
            if (length > 1) {
               int rankings[] = new int[length];
               int count = 0;
               int maxRanking = Integer.MIN_VALUE;

               for (int i = 0; i < length; i++) {
                  int ranking = ((ServiceReferenceImpl)
                     references[i]).getRanking();

                  rankings[i] = ranking;
```

```
            if (ranking > maxRanking) {
                index = i;
                maxRanking = ranking;
                count = 1;
            } else {
                if (ranking == maxRanking) {
                    count++;
                }
            }
        }
        //选择ID更低的。
        if (count > 1) {
            long minId = Long.MAX_VALUE;

            for (int i = 0; i < length; i++) {
                if (rankings[i] == maxRanking) {
                    long id = ((ServiceReferenceImpl)
                        references[i]).getId();

                    if (id < minId) {
                        index = i;
                        minId = id;
                    }
                }
            }
        }
    }
```

```
                return (references[index]);
            }
        } catch (InvalidSyntaxException e) {
        }

        return (null);
    }
    //获取服务对象和返回服务对象。
    public Object getService(
        org.osgi.framework.ServiceReference reference) {
        checkValid();

        synchronized (contextLock) {
            if (servicesInUse == null)
                servicesInUse = new Hashtable(10);
        }

        ServiceRegistrationImpl registration =
            ((ServiceReferenceImpl) reference).registration;


    framework.checkGetServicePermission(registration.clazz
es);

        return
registration.getService(BundleContextImpl.this);
    }
    public boolean ungetService(
        org.osgi.framework.ServiceReference reference) {
```

```
        checkValid();

        ServiceRegistrationImpl registration =
            ((ServiceReferenceImpl) reference).registration;

        return
registration.ungetService(BundleContextImpl.this);
    }
    public File getDataFile(String filename) {
        checkValid();

        return (framework.getDataFile(bundle, filename));
    }
    //启动激活器。
    protected void start() throws BundleException {
        activator = bundle.loadBundleActivator();

        if (activator != null) {
            try {
                startActivator(activator);
            } catch (BundleException be) {
                activator = null;
                throw be;
            }
        }
    }

    protected void startActivator(
        final BundleActivator bundleActivator)
```

```
        throws BundleException {
        try {
            AccessController.doPrivileged(
                new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    if (bundleActivator != null) {

    bundleActivator.start(BundleContextImpl.this);
                    }
                    return null;
                }
            });
        } catch (Throwable t) {
            if (t instanceof PrivilegedActionException) {
                t        =        ((PrivilegedActionException)
t).getException();
            }

            String clazz = null;
            clazz = bundleActivator.getClass().getName();

            throw new BundleException(
                NLS.bind(Msg.BUNDLE_ACTIVATOR_EXCEPTION,
                new     Object[]     {clazz,     "start",     、
                bundle.getSymbolicName() == null ? "" +
                bundle.getBundleId()                          :
            bundle.getSymbolicName()}), t);
        } finally {
        }
```

```
    }
    //停止激活器。
    protected void stop() throws BundleException {
        try {
            AccessController.doPrivileged(
                new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    if (activator != null) {
                        activator.stop(BundleContextImpl.this);
                    }
                    return null;
                }
            });
        } catch (Throwable t) {
            if (t instanceof PrivilegedActionException) {
                t        =        ((PrivilegedActionException)
t).getException();
            }

            String clazz = (activator == null) ? "" :
                activator.getClass().getName();

            throw new BundleException(
                NLS.bind(Msg.BUNDLE_ACTIVATOR_EXCEPTION,
                new Object[] {clazz, "stop",
                    bundle.getSymbolicName() == null ? "" +
                    bundle.getBundleId()                    :
                bundle.getSymbolicName()}), t);
```

```
    } finally {
        activator = null;
    }
}
//获取注册的服务。对于没有权限访问的服务，需要删除。
protected ServiceReference[] getRegisteredServices() {
    ServiceReference[] services = null;

    synchronized (framework.serviceRegistry) {
        services = framework.serviceRegistry.
            lookupServiceReferences(this);
        if (services == null) {
            return null;
        }
        int removed = 0;
        for (int i = services.length - 1; i >= 0; i--) {
            ServiceReferenceImpl ref =
                (ServiceReferenceImpl) services[i];
            String[] classes = ref.getClasses();
            try {

framework.checkGetServicePermission(classes);
            } catch (SecurityException se) {
                services[i] = null;
                removed++;
            }
        }
        if (removed > 0) {
            ServiceReference[] temp = services;
```

```
            services = new ServiceReference[
               temp.length - removed];
            for (int i = temp.length - 1; i >= 0; i--) {
               if (temp[i] == null)
                  removed--;
               else
                  services[i - removed] = temp[i];
            }
         }
      }
      return (services);
   }
   protected ServiceReferenceImpl[] getServicesInUse() {
      if (servicesInUse == null) {
         return (null);
      }

      synchronized (servicesInUse) {
         int size = servicesInUse.size();

         if (size == 0) {
            return (null);
         }

         ServiceReferenceImpl[] references =
            new ServiceReferenceImpl[size];
         int refcount = 0;

         Enumeration refsEnum = servicesInUse.keys();
```

```
            for (int i = 0; i < size; i++) {
                ServiceReferenceImpl reference =
                    (ServiceReferenceImpl)
                refsEnum.nextElement();

                try {
                    framework.checkGetServicePermission(
                        reference.registration.clazzes);
                } catch (SecurityException se) {
                    continue;
                }

                references[refcount] = reference;
                refcount++;
            }

            if (refcount < size) {
                if (refcount == 0) {
                    return (null);
                }

                ServiceReferenceImpl[] refs = references;
                references                  =                  new
ServiceReferenceImpl[refcount];
                System.arraycopy(refs,   0,   references,   0,
refcount);
            }

            return (references);
        }
```

```
    }
    //派发事件。
    public void dispatchEvent(Object originalListener, Object
l, int action, Object object) {
        AbstractBundle tmpBundle = bundle;
        try {
            if (isValid()) {
                switch (action) {
                    case Framework.BUNDLEEVENT :
                    case Framework.BUNDLEEVENTSYNC : {
                      BundleListener listener = (BundleListener)
l;

                       BundleEvent event = (BundleEvent) object;
                       switch (event.getType()) {
                          case Framework.BATCHEVENT_BEGIN : {
                             if (listener instanceof
                                BatchBundleListener)
                                ((BatchBundleListener)
listener).
                                   batchBegin();
                             break;
                          }
                          case Framework.BATCHEVENT_END : {
                             if (listener instanceof
                                BatchBundleListener)
                                ((BatchBundleListener)
                                   listener).batchEnd();
                             break;
```

```
                    }
                    default : {
                        listener.bundleChanged(
                            (BundleEvent) object);
                    }
                }
                break;
            }

            case Framework.SERVICEEVENT : {
                ServiceEvent event = (ServiceEvent)
object;
                ServiceListener listener =
                    (ServiceListener) l;
                listener.serviceChanged(event);
                break;
            }

            case Framework.FRAMEWORKEVENT : {
                FrameworkListener listener =
                    (FrameworkListener) l;
                listener.frameworkEvent(
                    (FrameworkEvent) object);
                break;
            }
        }
    }
} catch (Throwable t) {
    //处理意外的异常。
```

```
            framework.adaptor.handleRuntimeError(t);
            publisherror: {
               if (action == Framework.FRAMEWORKEVENT) {
                  FrameworkEvent  event  =  (FrameworkEvent)
object;
                  if (event.getType() == FrameworkEvent.ERROR)
{
                     break publisherror; //避免无限制循环。
                  }
               }

               framework.publishFrameworkEvent(
                  FrameworkEvent.ERROR, tmpBundle, t);
            }
         }
      }


      protected boolean hasListenServicePermission(
         ServiceEvent event) {
         ProtectionDomain              domain              =
bundle.getProtectionDomain();

         if (domain != null) {
            ServiceReferenceImpl reference =
            (ServiceReferenceImpl)
         event.getServiceReference();

            String[] names = reference.getClasses();
```

```
            int len = names.length;

            for (int i = 0; i < len; i++) {
               if                          (domain.implies(new
ServicePermission(names[i],
                  ServicePermission.GET))) {
                  return true;
               }
            }

            return false;
         }

         return (true);
      }
      public   org.osgi.framework.Filter   createFilter(String
filter)
         throws InvalidSyntaxException {
         checkValid();

         return (new FilterImpl(filter));
      }

      protected void checkValid() {
         if (!isValid()) {
            throw new IllegalStateException(
               Msg.BUNDLE_CONTEXT_INVALID_EXCEPTION);
         }
      }
```

```
    protected boolean isValid() {
        return valid;
    }

    boolean isAssignableTo(ServiceReferenceImpl reference) {
        if (!scopeEvents)
            return true;
        String[] clazzes = reference.getClasses();
        for (int i = 0; i < clazzes.length; i++)
            if (!reference.isAssignableTo(bundle, clazzes[i]))
                return false;
        return true;
    }
}
```

3.1.7 BundleLoaderProxy

```
package org.eclipse.osgi.framework.internal.core;

import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import org.eclipse.osgi.framework.util.KeyedHashSet;
import org.eclipse.osgi.service.resolver.*;
import org.osgi.framework.*;
import org.osgi.service.packageadmin.RequiredBundle;

//为一个Bundle代理BundlerLoader对象。允许无需创建BundleLoader或
```

```
//BundlerClassLoader对象而连接Bundle的依赖。保持Bundle依赖关系。
public class BundleLoaderProxy implements RequiredBundle {
   //Loader。
   private BundleLoader loader;
   //所在Bundle。
   final private BundleHost bundle;
   final private BundleDescription description;
   //指示代理过期。
   private boolean stale = false;
   //缓存Bundle包源。
   final private KeyedHashSet pkgSources;

   public BundleLoaderProxy(BundleHost bundle,
      BundleDescription description) {
      this.bundle = bundle;
      this.description = description;
      this.pkgSources = new KeyedHashSet(false);
   }
   //获取Loader。
   synchronized BundleLoader getBundleLoader() {
      if (loader != null) //直接返回。
         return loader;
      if (bundle.isResolved()) { //Bundle必须经过解析。
         try {
            if (bundle.getBundleId() == 0) //系统Bundle。
               loader = new SystemBundleLoader(bundle,
this);
```

```
                else
                    loader = new BundleLoader(bundle, this);
            } catch (BundleException e) {
                bundle.framework.publishFrameworkEvent(
                    FrameworkEvent.ERROR, bundle, e);
                return null;
            }
        }
        return loader;
    }
    //直接返回Loader，不会尝试创建。
    BundleLoader getBasicBundleLoader() {
        return loader;
    }
    AbstractBundle getBundleHost() {
        return bundle;
    }
    //设置代理过期。
    void setStale() {
        stale = true;
    }
    boolean isStale() {
        return stale;
    }

    public String toString() {
        String symbolicName = bundle.getSymbolicName();
        StringBuffer sb = new StringBuffer(symbolicName ==
null ?
```

```
            bundle.getBundleData().getLocation()                :
        symbolicName);
        sb.append(";
").append(Constants.BUNDLE_VERSION_ATTRIBUTE);
        sb.append("=\"").append(
            description.getVersion().toString()).append("\""
        );
        return sb.toString();
    }
    //如果过期的话，则返回空。
    public org.osgi.framework.Bundle getBundle() {
        if (isStale())
            return null;

        return bundle;
    }
  public org.osgi.framework.Bundle[] getRequiringBundles()
{
        if (isStale())
            return null;
        //这将很慢，因此不能在正常执行时调用。
        BundleDescription[] dependents =
            description.getDependents();
        if (dependents == null || dependents.length == 0)
            return null;
        ArrayList result = new ArrayList(dependents.length);
        for (int i = 0; i < dependents.length; i++)
            addRequirers(dependents[i], result);
        return result.size() == 0 ? null :
```

```
            (Bundle[]) result.toArray(
            new org.osgi.framework.Bundle[result.size()]);
    }

    void addRequirers(BundleDescription dependent,
        ArrayList result) {
        if (dependent.getHost() != null) //不管片段。
            return;
        BundleLoaderProxy dependentProxy =
            getBundleLoader().getLoaderProxy(dependent);
        if (dependentProxy == null)
            return; //Bundle已经被卸载。
        if (result.contains(dependentProxy.bundle))
            return; //阻止循环。
        BundleLoader dependentLoader =
            dependentProxy.getBundleLoader();
        BundleLoaderProxy[] requiredBundles =
            dependentLoader.requiredBundles;
        //返回Reexport表。
        int[] reexportTable = dependentLoader.reexportTable;
        if (requiredBundles == null)
            return;
        int size = reexportTable == null ? 0 :
reexportTable.length;
        int reexportIndex = 0;
        for (int i = 0; i < requiredBundles.length; i++) {
            if (requiredBundles[i] == this) {
                result.add(dependentProxy.bundle);
```

```
            if (reexportIndex < size &&
               reexportTable[reexportIndex] == i) {
               reexportIndex++;
               BundleDescription[] dependents =
                  dependent.getDependents();
               if (dependents == null)
                  return;
               for (int j = 0; j < dependents.length; j++)
                  dependentProxy.addRequirers(
                     dependents[j], result);
            }
            return;
         }
      }
      return;
   }

   public String getSymbolicName() {
      return description.getSymbolicName();
   }

   public Version getVersion() {
      return description.getVersion();
   }

   public boolean isRemovalPending() {
      return description.isRemovalPending();
   }

   BundleDescription getBundleDescription() {
```

```
        return description;
    }

    PackageSource getPackageSource(String pkgName) {
        PackageSource pkgSource =
            (PackageSource) pkgSources.getByKey(pkgName);
        if (pkgSource == null) {
            pkgSource = new SingleSourcePackage(pkgName, -1,
this);
            synchronized (pkgSources) {
                pkgSources.add(pkgSource);
            }
        }
        return pkgSource;
    }

    boolean inUse() {
        return description.getDependents().length > 0;
    }

    boolean    forceSourceCreation(ExportPackageDescription
export) {
        if (!export.isRoot())
            return true;
        boolean strict = Constants.STRICT_MODE.equals(
            bundle.framework.adaptor.getState().
            getPlatformProperties()[0].get(
            Constants.OSGI_RESOLVER_MODE));
        return (export.getDirective(
            Constants.INCLUDE_DIRECTIVE) !=
```

```
            null) || (export.getDirective(
            Constants.EXCLUDE_DIRECTIVE) != null) ||
            (strict && export.getDirective(
            Constants.FRIENDS_DIRECTIVE) != null);
    }

    PackageSource createPackageSource(
        ExportPackageDescription export, boolean storeSource)
    {
        PackageSource pkgSource = null;

        if (!export.isRoot()) {
            pkgSource                    =                    new
ReexportPackageSource(export.getName());
        } else {
            String includes = (String) export.getDirective(
                Constants.INCLUDE_DIRECTIVE);
            String excludes = (String) export.getDirective(
                Constants.EXCLUDE_DIRECTIVE);
            String[] friends = (String[]) export.getDirective(
                Constants.FRIENDS_DIRECTIVE);
            if (friends != null) {
                boolean strict = Constants.STRICT_MODE.equals(
                    bundle.framework.adaptor.getState().
                    getPlatformProperties()[0].get(
                    Constants.OSGI_RESOLVER_MODE));
                if (!strict)
                    friends = null;
            }
            if (includes != null || excludes != null ||
```

```
            friends != null) {
            ExportPackageDescription[] exports
               = description.getExportPackages();
            int index = -1;
            int first = -1;
            for (int i = 0; i < exports.length; i++) {
               if (first == -1 &&
                  exports[i].getName().equals(export.getNa
               me()))
                  first = i;
               if (exports[i] == export && first != i) {
                  index = i;
                  break;
               }
            }
            pkgSource = new FilteredSourcePackage(
               export.getName(),  index,  this,  includes,
            excludes,
               friends);
         }
      }

      if (storeSource) {

         if (pkgSource != null &&
            pkgSources.getByKey(export.getName()) == null)
            synchronized (pkgSources) {
               pkgSources.add(pkgSource);
            }
      } else {
```

```
            if (pkgSource == null)
                pkgSource = getPackageSource(export.getName());
        }

        return pkgSource;
    }

    class ReexportPackageSource extends PackageSource {
        public ReexportPackageSource(String id) {
            super(id);
        }

        public synchronized SingleSourcePackage[]
getSuppliers() {
            PackageSource source =
                getBundleLoader().getPackageSource(id);
            if (source == null)
                return null;
            return source.getSuppliers();
        }

        public Class loadClass(String name) {
            try {
                return getBundleLoader().findClass(name,
false);
            } catch (ClassNotFoundException e) {
                return null;
            }
        }
```

```
        public URL getResource(String name) {
            return        getBundleLoader().findResource(name,
false);
        }

        public Enumeration getResources(String name)
        throws IOException {
            return getBundleLoader().findResources(name);
        }
    }
}
```

3.1.8 BundleLoader

```
package org.eclipse.osgi.framework.internal.core;

import java.io.IOException;
import java.net.URL;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.*;
import org.eclipse.osgi.framework.adaptor.*;
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.framework.util.KeyedHashSet;
import org.eclipse.osgi.service.resolver.*;
import org.eclipse.osgi.util.ManifestElement;
import org.osgi.framework.BundleException;
import org.osgi.framework.FrameworkEvent;

//这个对象负责一个Bundle的类加载代理。它表示Bundle的装载状态。
```

BundleLoader

//对象采用懒创建方式创建。除非必要，否则不要强迫创建一个

BundleLoader。

```
public class BundleLoader implements ClassLoaderDelegate {
    public final static String DEFAULT_PACKAGE = ".";
    public final static String JAVA_PACKAGE = "java.";
    public final static byte FLAG_IMPORTSINIT = 0x01;
    public final static byte FLAG_HASDYNAMICIMPORTS = 0x02;
    public final static byte FLAG_HASDYNAMICEIMPORTALL = 0x04;
    public final static byte FLAG_CLOSED = 0x08;

    public final static ClassContext CLASS_CONTEXT =
    (ClassContext) AccessController.doPrivileged(
        new PrivilegedAction() {
        public Object run() {
            return new ClassContext();
        }
    });
    public final static ClassLoader FW_CLASSLOADER =
        getClassLoader(Framework.class);

    private static final boolean USE_GLOBAL_DEADLOCK_AVOIDANCE_LOCK =
        "true".equals(FrameworkProperties.getProperty(
            "osgi.classloader.singleThreadLoads"));
    private static final List waitingList =
        USE_GLOBAL_DEADLOCK_AVOIDANCE_LOCK ? new ArrayList(0) : null;
    private static Object lockThread;
    private static int lockCount = 0;
```

```
//Proxy。
final private BundleLoaderProxy proxy;
//所在Bundle。
final BundleHost bundle;
final private PolicyHandler policy;
//导出包。
final private Collection exportedPackages;
//必须Bundle。
final BundleLoaderProxy[] requiredBundles;
//Re-export表。
final int[] reexportTable;
//必须包源缓存，【包名，PackageSource】。
final private KeyedHashSet requiredSources;
//导入包缓存。
private KeyedHashSet importedSources;
private String[] dynamicImportPackageStems;
private String[] dynamicImportPackages;
//装载标记。
private byte loaderFlags = 0;
//这个Bundle的类加载器。
private BundleClassLoader classloader;
//父类加载器。
private ClassLoader parent;


//返回指定类名的包名。
public final static String getPackageName(String name) {
```

```
    if (name != null) {
        int index = name.lastIndexOf('.'); //最后一个“.”之前部分。
        if (index > 0)
            return name.substring(0, index);
    }
    return DEFAULT_PACKAGE;
}

//从指定的资源名称返回其包名。资源的名称由“/”分割。
public final static String getResourcePackageName(String name) {
    if (name != null) {
        int begin = ((name.length() > 1) &&
            (name.charAt(0) == '/')) ? 1 : 0;
        int end = name.lastIndexOf('/');
        if (end > begin)
            return name.substring(begin, end).replace('/', '.');
    }
    return DEFAULT_PACKAGE;
}

//构造器，这个对象在对该Bundle第一次资源请求时创建。
//创建时将获取这个Bundle的描述信息，并初始化
protected BundleLoader(BundleHost bundle, BundleLoaderProxy proxy)
    throws BundleException {
    this.bundle = bundle;
    this.proxy = proxy;
    try {
        bundle.getBundleData().open(); //确保BundleData打开。
```

```
        } catch (IOException e) {
            throw new BundleException(Msg.BUNDLE_READ_EXCEPTION, e);
        }
        //Bundle描述信息。
        BundleDescription description = proxy.getBundleDescription();
        //Required描述信息。
        BundleDescription[] required =
            description.getResolvedRequires();
        //初始化Re-export表，RequireBundles信息。
        if (required.length > 0) {
            HashSet reExportSet = new HashSet(required.length);
            BundleSpecification[] requiredSpecs =
                description.getRequiredBundles();
            if (requiredSpecs != null && requiredSpecs.length > 0)
                for (int i = 0; i < requiredSpecs.length; i++)
                    if (requiredSpecs[i].isExported())
                        reExportSet.add(requiredSpecs[i].getName());

            requiredBundles = new BundleLoaderProxy[required.length];
            int[] reexported = new int[required.length];
            int reexportIndex = 0;
            for (int i = 0; i < required.length; i++) {
                requiredBundles[i] = getLoaderProxy(required[i]);
                if
(reExportSet.contains(required[i].getSymbolicName()))
                    reexported[reexportIndex++] = i;
            }
            if (reexportIndex > 0) {
                reexportTable = new int[reexportIndex];
```

```
            System.arraycopy(
                reexported, 0, reexportTable, 0, reexportIndex);
        } else {
            reexportTable = null;
        }
        requiredSources = new KeyedHashSet(10, false);
    } else {
        requiredBundles = null;
        reexportTable = null;
        requiredSources = null;
    }

    //初始化导出包信息。
    ExportPackageDescription[] exports =
        description.getSelectedExports();
    if (exports != null && exports.length > 0) {
        exportedPackages = exports.length > 10 ?
            (Collection) new HashSet(exports.length) :
            new ArrayList(exports.length);
        for (int i = 0; i < exports.length; i++) {
            if (proxy.forceSourceCreation(exports[i])) {
                if (!exportedPackages.contains(
                    exports[i].getName())) {
                    //必须强迫过滤和早创建Re-export源来阻止晚正常
                    包源
                    //创建。在第一个Export是做。
                    proxy.createPackageSource(exports[i], true);
                }
            }
```

```
            exportedPackages.add(exports[i].getName());
        }
    } else {
        exportedPackages = null;
    }
    //初始化片段。
    org.osgi.framework.Bundle[] fragmentObjects =
        bundle.getFragments();
    BundleDescription[] fragments = new BundleDescription[
        fragmentObjects == null ? 0 : fragmentObjects.length];
    for (int i = 0; i < fragments.length; i++)
        fragments[i] = ((AbstractBundle) fragmentObjects[i]).
            getBundleDescription();

    if (description.hasDynamicImports())
        addDynamicImportPackage(description.getImportPackages());

    for (int i = 0; i < fragments.length; i++)
        if (fragments[i].isResolved() &&
            fragments[i].hasDynamicImports())
            addDynamicImportPackage(
                fragments[i].getImportPackages());

    //初始化PolicyHandler。
    String buddyList = null;
    try {
        buddyList = (String) bundle.getBundleData().
            getManifest().get(Constants.BUDDY_LOADER);
    } catch (BundleException e) {
```

```
        }
        policy = buddyList != null ?
            new PolicyHandler(this, buddyList) : null;
    }

    private synchronized KeyedHashSet getImportedSources() {
        if ((loaderFlags & FLAG_IMPORTSINIT) != 0)
            return importedSources;
        ExportPackageDescription[] packages =
            proxy.getBundleDescription().getResolvedImports();
        if (packages != null && packages.length > 0) {
            if (importedSources == null)
                importedSources = new KeyedHashSet(
                    packages.length, false);
            for (int i = 0; i < packages.length; i++) {
                PackageSource source =
                    createExportPackageSource(packages[i]);
                if (source != null)
                    importedSources.add(source);
            }
        }
        loaderFlags |= FLAG_IMPORTSINIT;
        return importedSources;
    }

    final PackageSource createExportPackageSource(
        ExportPackageDescription export) {
        BundleLoaderProxy exportProxy =
            getLoaderProxy(export.getExporter());
        if (exportProxy == null)
```

```
            return null;
        PackageSource requiredSource = exportProxy.getBundleLoader().
            findRequiredSource(export.getName());
        PackageSource exportSource = exportProxy.createPackageSource(
            export, false);
        if (requiredSource == null)
            return exportSource;
        return createMultiSource(export.getName(),
            new PackageSource[] {requiredSource, exportSource});
    }

    private static PackageSource createMultiSource(
        String packageName, PackageSource[] sources) {
        if (sources.length == 1)
            return sources[0];
        ArrayList sourceList = new ArrayList(sources.length);
        for (int i = 0; i < sources.length; i++) {
            SingleSourcePackage[] innerSources =
                sources[i].getSuppliers();
            for (int j = 0; j < innerSources.length; j++)
                if (!sourceList.contains(innerSources[j]))
                    sourceList.add(innerSources[j]);
        }
        return new MultiSourcePackage(packageName,
            (SingleSourcePackage[]) sourceList.toArray(
            new SingleSourcePackage[sourceList.size()]));
    }
```

```
    //获取对应的BundleLoaderProxy。
    final BundleLoaderProxy getLoaderProxy(BundleDescription source) {
        BundleLoaderProxy sourceProxy = (BundleLoaderProxy)
source.getUserObject();
        if (sourceProxy == null) {
            long exportingID = source.getBundleId();
            BundleHost exportingBundle =
                (BundleHost) bundle.framework.getBundle(exportingID);
            if (exportingBundle == null)
                return null;
            sourceProxy = exportingBundle.getLoaderProxy();
        }
        return sourceProxy;
    }

    //关闭。
    synchronized void close() {
        if ((loaderFlags & FLAG_CLOSED) != 0)
            return;
        if (classloader != null)
            classloader.close();
        if (policy != null)
            policy.close();
        loaderFlags |= FLAG_CLOSED;
    }

    //装载类。从宿主、片段、Import、Require和本地资源搜索。
    final Class loadClass(String name) throws ClassNotFoundException
    {
```

```
    return createClassLoader().loadClass(name);
}

//搜索资源。
final URL getResource(String name) {
    return createClassLoader().getResource(name);
}

final synchronized ClassLoader getParentClassLoader() {
    if (parent != null)
        return parent;
    createClassLoader();
    return parent;
}
//创建类加载器。
final synchronized BundleClassLoader createClassLoader() {
    if (classloader != null)
        return classloader;
    try {
        String[] classpath = bundle.getBundleData().getClassPath();
        if (classpath != null) {
            BundleClassLoader bcl = createBCLPrevileged(
                bundle.getProtectionDomain(), classpath);
            parent = getParentPrivileged(bcl);
            classloader = bcl;
        } else {
            bundle.framework.publishFrameworkEvent(
                FrameworkEvent.ERROR, bundle, new BundleException(
                Msg.BUNDLE_NO_CLASSPATH_MATCH));
```

```
            }
        } catch (BundleException e) {
            bundle.framework.publishFrameworkEvent(
                FrameworkEvent.ERROR, bundle, e);
        }
        return classloader;
    }
    //查找Bundle本地的类。仅搜索这个Bundle的类加载器。
    Class findLocalClass(String name) throws ClassNotFoundException {
        try {
            Class clazz = createClassLoader().findLocalClass(name);
            return clazz;
        } catch (ClassNotFoundException e) {
            if (e instanceof StatusException) {
                if ((((StatusException) e).getStatusCode() &
                    StatusException.CODE_ERROR) != 0)
                    throw e;
            }
            return null;
        }
    }

    //查找一个Bundle的类。
    public Class findClass(String name) throws ClassNotFoundException
    {
        return findClass(name, true);
    }

    Class findClass(String name, boolean checkParent)
```

```
        throws ClassNotFoundException {
        ClassLoader parentCL = getParentClassLoader();
        //如果checkParent为真，父类加载器不为空且类名从java起。
        if (checkParent && parentCL != null &&
            name.startsWith(JAVA_PACKAGE))
            // 1) 搜索第一步，如果该类类名由“java.”字符开始，则代理到父类。
            return parentCL.loadClass(name);
        try {
            if (USE_GLOBAL_DEADLOCK_AVOIDANCE_LOCK)
                lock(createClassLoader());
            return findClassInternal(name, checkParent, parentCL);
        } finally {
            if (USE_GLOBAL_DEADLOCK_AVOIDANCE_LOCK)
                unlock();
        }
    }

    private Class findClassInternal(String name, boolean checkParent,
ClassLoader parentCL) throws ClassNotFoundException {
        if (Debug.DEBUG && Debug.DEBUG_LOADER)
            Debug.println("BundleLoader[" + this + "].loadBundleClass("
+ name + ")");  //$NON-NLS-3$
        String pkgName = getPackageName(name);
        boolean bootDelegation = false;
        // 遵循OSGi代理模型。
        if (checkParent && parentCL != null &&
isBootDelegationPackage(pkgName))
            // 2) 如果是启动代理列表的一部分，则代理到父类加载器。
            try {
```

```
            return parentCL.loadClass(name);
        } catch (ClassNotFoundException cnfe) {
            //继续。
            bootDelegation = true;
        }

Class result = null;
// 3) 搜索Import。
PackageSource source = findImportedSource(pkgName);
if (source != null) { //如果该类在Import中。
    // 3) 查找Import源，终止搜索。
    result = source.loadClass(name);
    if (result != null)
        return result;
    //抛出异常。
    throw new ClassNotFoundException(name);
}
// 4) 搜索RequireBundles。
source = findRequiredSource(pkgName);
if (source != null)
    // 4) 尝试加载但失败时继续。
    result = source.loadClass(name);
// 5) 搜索本地。
if (result == null)
    result = findLocalClass(name);
if (result != null)
    return result;
// 6) 尝试查找动态引用。
if (source == null) {
```

```
            source = findDynamicSource(pkgName);
            if (source != null) {
                result = source.loadClass(name);
                if (result != null)
                    return result;
                //如果在动态引用列表却加载失败，则抛出异常。
                throw new ClassNotFoundException(name);
            }
        }

        // 进行Buddy类加载策略。
        if (result == null && policy != null)
            result = policy.doBuddyClassLoading(name);
        // 向后兼容
        if (parentCL != null && checkParent && !bootDelegation &&
            bundle.framework.compatibiltyBootDelegation &&
            result == null && source == null &&
             !isExportedPackage(pkgName))
            return parentCL.loadClass(name);
            if (parentCL != null && result == null &&
                !bootDelegation && isRequestFromVM())
            result = parentCL.loadClass(name);
        if (result == null)
            throw new ClassNotFoundException(name);
        return result;
    }

    private boolean isRequestFromVM() {
        if (bundle.framework.bootDelegateAll ||
            !bundle.framework.contextBootDelegation)
```

```
            return false;


        Class[] context = CLASS_CONTEXT.getClassContext();
        if (context == null || context.length < 2)
            return false;
        // 第一个类是ClassContext，忽略。
        for (int i = 1; i < context.length; i++)
            // 查找第一个不是BundleLoader或ClassLoader实例的Context。
        if (context[i] != BundleLoader.class &&
             !ClassLoader.class.isAssignableFrom(context[i])) {
                // 仅从Parent找。
                ClassLoader cl = getClassLoader(context[i]);
                if (cl != FW_CLASSLOADER) {
                    if (Class.class != context[i] &&
                        !(cl instanceof BundleClassLoader))
                        return true;
                    break;
                }
            }
        return false;
    }
    //获取类加载器。
    private static ClassLoader getClassLoader(final Class clazz) {
        if (System.getSecurityManager() == null)
            return clazz.getClassLoader();
        return (ClassLoader) AccessController.doPrivileged(
            new PrivilegedAction() {
            public Object run() {
                return clazz.getClassLoader();
```

```
        }
    });
}


//查找资源。
public URL findResource(String name) {
    return findResource(name, true);
}

URL findResource(String name, boolean checkParent) {
    //删除第一个'/'字符。
    if ((name.length() > 1) && (name.charAt(0) == '/'))
        name = name.substring(1);
    //获取包源。
    String pkgName = getResourcePackageName(name);
    boolean bootDelegation = false;
    ClassLoader parentCL = getParentClassLoader();
    //遵循OSGi搜索模型。
    //首先为系统资源检查父类加载器。
    if (checkParent && parentCL != null) {
        if (pkgName.startsWith(JAVA_PACKAGE))
            // 1) 如果资源名称以'java.'开头，则交给父加载器其并终止搜索。
            return parentCL.getResource(name);
        else if (isBootDelegationPackage(pkgName)) {
            // 2) 如果是启动代理列表。
            URL result = parentCL.getResource(name);
            if (result != null) //若失败，继续。
```

```
            return result;
        bootDelegation = true;
    }
}

URL result = null;
// 3) 搜索Import。
PackageSource source = findImportedSource(pkgName);
if (source != null)
    // 3) 在Import源搜索并终止。
    return source.getResource(name);
// 4) 搜索必须Bundle。
source = findRequiredSource(pkgName);
if (source != null)
    // 4) 尝试，若失败继续。
    result = source.getResource(name);
// 5) 搜索本地。
if (result == null)
    result = findLocalResource(name);
if (result != null)
    return result;
// 6) 动态引用。
if (source == null) {
    source = findDynamicSource(pkgName);
    if (source != null)
        //返回并终止搜索。
        return source.getResource(name);
}

//兄弟策略状态。
```

```
    if (result == null && policy != null)
        result = policy.doBuddyResourceLoading(name);
    //向后兼容。
    if (parentCL != null && checkParent && !bootDelegation &&
        bundle.framework.compatibiltyBootDelegation &&
        result == null && source == null &
        !isExportedPackage(pkgName))
            return parentCL.getResource(name);

    if (parentCL != null && result == null && !bootDelegation &&
        isRequestFromVM())
        result = parentCL.getResource(name);
    return result;
}
//判断是否为启动代理的包。
boolean isBootDelegationPackage(String name) {
    if (bundle.framework.bootDelegateAll)
        return true;
    if (bundle.framework.bootDelegation != null)
        for (int i = 0; i < bundle.framework.bootDelegation.length;
            i++)
            if (name.equals(bundle.framework.bootDelegation[i]))
                return true;
    if (bundle.framework.bootDelegationStems != null)
        for (int i = 0;
            i < bundle.framework.bootDelegationStems.length; i++)
            if (name.startsWith(
                bundle.framework.bootDelegationStems[i]))
                return true;
```

```
        return false;
    }

    //查找一个Bundle的资源。仅代理到Bundle的类加载器。
    public Enumeration findResources(String name) throws IOException
    {
        //不能代理到父加载器。
        if ((name.length() > 1) && (name.charAt(0) == '/'))
            name = name.substring(1);
        //获取资源包名。
        String pkgName = getResourcePackageName(name);
        Enumeration result = null;
        // 从第三步开始，因为仅在Bundle类加载器找。
        // 3) 从Import找。
        PackageSource source = findImportedSource(pkgName);
        if (source != null)
            // 3) 终止搜索。
            return source.getResources(name);
        // 4) 搜索Require。
        source = findRequiredSource(pkgName);
        if (source != null)
            // 4) 继续。
            result = source.getResources(name);

        // 5) 搜索本地。
        // 组合本地。
        Enumeration localResults = findLocalResources(name);
        result = compoundEnumerations(result, localResults);
        // 6) 动态Import。
```

```
    if (result == null && source == null) {
        source = findDynamicSource(pkgName);
        if (source != null)
            return source.getResources(name);
    }
    //兄弟加载策略。
    if (policy != null) {
        Enumeration buddyResult =
            policy.doBuddyResourcesLoading(name);
        result = compoundEnumerations(result, buddyResult);
    }
    return result;
}

//遵循OSGi模型搜索。
Enumeration getResources(String name) throws IOException {
    if ((name.length() > 1) && (name.charAt(0) == '/'))
        name = name.substring(1);
    String pkgName = getResourcePackageName(name);

    Enumeration result = null;
    if (pkgName.startsWith(JAVA_PACKAGE) ||
        isBootDelegationPackage(pkgName)) {

        ClassLoader parentCL = getParentClassLoader();
        result = parentCL == null ? null : parentCL.getResources(name);
        if (pkgName.startsWith(JAVA_PACKAGE))
            return result;
    }
```

```
    return compoundEnumerations(result, findResources(name));
}

static Enumeration compoundEnumerations(
    Enumeration list1, Enumeration list2) {
    if (list2 == null || !list2.hasMoreElements())
        return list1;
    if (list1 == null || !list1.hasMoreElements())
        return list2;
    Vector compoundResults = new Vector();
    while (list1.hasMoreElements())
        compoundResults.add(list1.nextElement());
    while (list2.hasMoreElements()) {
        Object item = list2.nextElement();
        if (!compoundResults.contains(item))
            compoundResults.add(item);
    }
    return compoundResults.elements();
}

URL findLocalResource(final String name) {
    return createClassLoader().findLocalResource(name);
}

Enumeration findLocalResources(String name) {
    return createClassLoader().findLocalResources(name);
}

//查找Native库。
```

```
    public String findLibrary(final String name) {
        if (System.getSecurityManager() == null)
            return findLocalLibrary(name);
        return (String) AccessController.doPrivileged(
            new PrivilegedAction() {
            public Object run() {
                return findLocalLibrary(name);
            }
        });
    }

    final String findLocalLibrary(final String name) {
        String result = bundle.getBundleData().findLibrary(name);
        if (result != null)
            return result;

        org.osgi.framework.Bundle[] fragments = bundle.getFragments();
        if (fragments == null || fragments.length == 0)
            return null;

        //查找片段Import。
        for (int i = 0; i < fragments.length; i++) {
            result = ((AbstractBundle)
fragments[i]).getBundleData().findLibrary(name);
            if (result != null)
                return result;
        }
        return result;
    }
```

```
final AbstractBundle getBundle() {
    return bundle;
}

private BundleClassLoader createBCLPrevileged(
    final BundleProtectionDomain pd, final String[] cp) {
    if (System.getSecurityManager() == null)
        return createBCL(pd, cp);

    return (BundleClassLoader) AccessController.doPrivileged(
        new PrivilegedAction() {
        public Object run() {
            return createBCL(pd, cp);
        }
    });
}
//创建Bundle类加载器器。
BundleClassLoader createBCL(
    final BundleProtectionDomain pd, final String[] cp) {
    BundleClassLoader bcl = bundle.getBundleData().
        createClassLoader(BundleLoader.this, pd, cp);

    org.osgi.framework.Bundle[] fragments = bundle.getFragments();
    if (fragments != null)
        for (int i = 0; i < fragments.length; i++) {
            AbstractBundle fragment = (AbstractBundle) fragments[i];
            try {
                bcl.attachFragment(fragment.getBundleData(),
                    fragment.domain,
```

```
                    fragment.getBundleData().getClassPath());
            } catch (BundleException be) {
                bundle.framework.publishFrameworkEvent(
                    FrameworkEvent.ERROR, bundle, be);
            }
        }

    //初始化。
    bcl.initialize();
    return bcl;
}

public final String toString() {
    BundleData result = bundle.getBundleData();
    return result == null ? "BundleLoader.bundledata == null!" :
        result.toString();
}

private final synchronized boolean isDynamicallyImported(
    String pkgname) {
    if (this instanceof SystemBundleLoader)
        return false;
    //R3规范。
    if (pkgname.startsWith("java."))
        return true;

    if ((loaderFlags & FLAG_HASDYNAMICIMPORTS) == 0)
        return false;

    if ((loaderFlags & FLAG_HASDYNAMICEIMPORTALL) != 0)
```

```
            return true;

        if (dynamicImportPackages != null)
            for (int i = 0; i < dynamicImportPackages.length; i++)
                if (pkgname.equals(dynamicImportPackages[i]))
                    return true;

        if (dynamicImportPackageStems != null)
            for (int i = 0; i < dynamicImportPackageStems.length; i++)
                if (pkgname.startsWith(dynamicImportPackageStems[i]))
                    return true;

        return false;
    }

    final void addExportedProvidersFor(String symbolicName,
        String packageName, ArrayList result, KeyedHashSet visited) {
        if (!visited.add(bundle))
            return;

        // See if we locally provide the package.
        PackageSource local = null;
        if (isExportedPackage(packageName))
            local = proxy.getPackageSource(packageName);
        // Must search required bundles that are exported first.
        if (requiredBundles != null) {
            int size = reexportTable == null ? 0 : reexportTable.length;
            int reexportIndex = 0;
            for (int i = 0; i < requiredBundles.length; i++) {
                if (local != null) {
```

```
                    // always add required bundles first if we locally
provide the package
                    // This allows a bundle to provide a package from a
required bundle without
                    // re-exporting the whole required bundle.

    requiredBundles[i].getBundleLoader().addExportedProvidersFor(symb
olicName, packageName, result, visited);
                } else if (reexportIndex < size &&
reexportTable[reexportIndex] == i) {
                    reexportIndex++;

    requiredBundles[i].getBundleLoader().addExportedProvidersFor(symb
olicName, packageName, result, visited);
                }
            }
        }

        // now add the locally provided package.
        if (local != null && local.isFriend(symbolicName)) {
            if (local instanceof
BundleLoaderProxy.ReexportPackageSource)
                local = new SingleSourcePackage(packageName, -1, proxy);
            result.add(local);
        }
    }

    final boolean isExportedPackage(String name) {
        return exportedPackages == null ? false :
exportedPackages.contains(name);
```

```
    }

    private void addDynamicImportPackage(ImportPackageSpecification[]
packages) {
        if (packages == null)
            return;
        ArrayList dynamicImports = new ArrayList(packages.length);
        for (int i = 0; i < packages.length; i++)
            if
(ImportPackageSpecification.RESOLUTION_DYNAMIC.equals(packages[i].get
Directive(Constants.RESOLUTION_DIRECTIVE)))
                dynamicImports.add(packages[i].getName());
        if (dynamicImports.size() > 0)
            addDynamicImportPackage((String[])
dynamicImports.toArray(new String[dynamicImports.size()]));
    }

    /**
     * Adds a list of DynamicImport-Package manifest elements to the
dynamic
     * import tables of this BundleLoader.  Duplicate packages are checked
and
     * not added again.  This method is not thread safe.  Callers should
ensure
     * synchronization when calling this method.
     * @param packages the DynamicImport-Package elements to add.
     */
    private void addDynamicImportPackage(String[] packages) {
        if (packages == null)
            return;
```

```
        loaderFlags |= FLAG_HASDYNAMICIMPORTS;
        int size = packages.length;
        ArrayList stems;
        if (dynamicImportPackageStems == null) {
            stems = new ArrayList(size);
        } else {
            stems = new ArrayList(size +
dynamicImportPackageStems.length);
            for (int i = 0; i < dynamicImportPackageStems.length; i++) {
                stems.add(dynamicImportPackageStems[i]);
            }
        }

        ArrayList names;
        if (dynamicImportPackages == null) {
            names = new ArrayList(size);
        } else {
            names = new ArrayList(size + dynamicImportPackages.length);
            for (int i = 0; i < dynamicImportPackages.length; i++) {
                names.add(dynamicImportPackages[i]);
            }
        }

        for (int i = 0; i < size; i++) {
            String name = packages[i];
            if (isDynamicallyImported(name))
                continue;
            if (name.equals("*")) { /* shortcut *///$NON-NLS-1$
                loaderFlags |= FLAG_HASDYNAMICEIMPORTALL;
```

```
                return;
            }

            if (name.endsWith(".*"))
                stems.add(name.substring(0, name.length() - 1));
            else
                names.add(name);
        }

        size = stems.size();
        if (size > 0)
            dynamicImportPackageStems = (String[]) stems.toArray(new
String[size]);

        size = names.size();
        if (size > 0)
            dynamicImportPackages = (String[]) names.toArray(new
String[size]);
    }

    /**
     * Adds a list of DynamicImport-Package manifest elements to the
dynamic
     * import tables of this BundleLoader.  Duplicate packages are checked
and
     * not added again.
     * @param packages the DynamicImport-Package elements to add.
     */
    public final synchronized void
addDynamicImportPackage(ManifestElement[] packages) {
```

```
        if (packages == null)
            return;
        ArrayList dynamicImports = new ArrayList(packages.length);
        for (int i = 0; i < packages.length; i++)
            dynamicImports.add(packages[i].getValue());
        if (dynamicImports.size() > 0)
            addDynamicImportPackage((String[])
dynamicImports.toArray(new String[dynamicImports.size()]));
    }

    final synchronized void attachFragment(BundleFragment fragment)
throws BundleException {
        if (classloader == null)
            return;
        String[] classpath = fragment.getBundleData().getClassPath();
        if (classpath != null)
            classloader.attachFragment(fragment.getBundleData(),
fragment.domain, classpath);
    }

    /*
     * Finds a packagesource that is either imported or required from
another bundle.
     * This will not include an local package source
     */
    private PackageSource findSource(String pkgName) {
        if (pkgName == null)
            return null;
        PackageSource result = findImportedSource(pkgName);
        if (result != null)
```

```
            return result;
        // Note that dynamic imports are not checked to avoid aggressive
wiring (bug 105779)
        return findRequiredSource(pkgName);
    }

    private PackageSource findImportedSource(String pkgName) {
        KeyedHashSet imports = getImportedSources();
        if (imports == null)
            return null;
        synchronized (imports) {
            return (PackageSource) imports.getByKey(pkgName);
        }
    }

    private PackageSource findDynamicSource(String pkgName) {
        if (isDynamicallyImported(pkgName)) {
            ExportPackageDescription exportPackage =
bundle.framework.adaptor.getState().linkDynamicImport(proxy.getBundle
Description(), pkgName);
            if (exportPackage != null) {
                PackageSource source =
createExportPackageSource(exportPackage);
                synchronized (this) {
                    if (importedSources == null)
                        importedSources = new KeyedHashSet(false);
                }
                synchronized (importedSources) {
                    importedSources.add(source);
                }
```

```
                return source;
            }
        }
        return null;
    }

    private PackageSource findRequiredSource(String pkgName) {
        if (requiredBundles == null)
            return null;
        synchronized (requiredSources) {
            PackageSource result = (PackageSource)
requiredSources.getByKey(pkgName);
            if (result != null)
                return result.isNullSource() ? null : result;
        }
        KeyedHashSet visited = new KeyedHashSet(false);
        visited.add(bundle); // always add ourselves so we do not recurse
back to ourselves
        ArrayList result = new ArrayList(3);
        for (int i = 0; i < requiredBundles.length; i++) {
            BundleLoader requiredLoader =
requiredBundles[i].getBundleLoader();

    requiredLoader.addExportedProvidersFor(proxy.getSymbolicName(),
pkgName, result, visited);
        }
        // found some so cache the result for next time and return
        PackageSource source;
        if (result.size() == 0) {
            // did not find it in our required bundles lets record the
```

```
failure
            // so we do not have to do the search again for this package.
            source = NullPackageSource.getNullPackageSource(pkgName);
        } else if (result.size() == 1) {
            // if there is just one source, remember just the single source
            source = (PackageSource) result.get(0);
        } else {
            // if there was more than one source, build a multisource and
cache that.
            PackageSource[] srcs = (PackageSource[]) result.toArray(new
PackageSource[result.size()]);
            source = createMultiSource(pkgName, srcs);
        }
        synchronized (requiredSources) {
            requiredSources.add(source);
        }
        return source.isNullSource() ? null : source;
    }

    /*
     * Gets the package source for the pkgName.  This will include the local
package source
     * if the bundle exports the package.  This is used to compare the
PackageSource of a
     * package from two different bundles.
     */
    final PackageSource getPackageSource(String pkgName) {
        PackageSource result = findSource(pkgName);
        if (!isExportedPackage(pkgName))
            return result;
```

```
        // if the package is exported then we need to get the local source
        PackageSource localSource = proxy.getPackageSource(pkgName);
        if (localSource instanceof
BundleLoaderProxy.ReexportPackageSource)
            localSource = new SingleSourcePackage(pkgName, -1, proxy);
        if (result == null)
            return localSource;
        if (localSource == null)
            return result;
        return createMultiSource(pkgName, new PackageSource[] {result,
localSource});
    }

    private ClassLoader getParentPrivileged(final BundleClassLoader bcl)
{
        if (System.getSecurityManager() == null)
            return bcl.getParent();

        return (ClassLoader) AccessController.doPrivileged(new
PrivilegedAction() {
            public Object run() {
                return bcl.getParent();
            }
        });
    }

    static final class ClassContext extends SecurityManager {
        // need to make this method public
        public Class[] getClassContext() {
            return super.getClassContext();
```

```
        }
    }

    /*
     * see bug 121737
     * To ensure that we do not enter a deadly embrace between classloader
cycles
     * we attempt to obtain a global lock before do normal osgi delegation.
     * This approach ensures that only one thread has a classloader locked
at a time
     */
    private static void lock(Object loader) {
        Thread currentThread = Thread.currentThread();
        boolean interrupted = false;
        synchronized (loader) {
            if (tryLock(currentThread, loader))
                return; // this thread has the lock

            do {
                try {
                    // we wait on the loader object here to release its lock
incase we have it.
                    // we do not way to wait while holding this lock because
that will cause deadlock
                    loader.wait();
                } catch (InterruptedException e) {
                    interrupted = true;
                    // we still want to try again
                }
            } while (!tryLock(currentThread));
```

```
        }
        if (interrupted)
            currentThread.interrupt();
    }

    /*
     * returns true if this thread can obtain the global lock or already
has the lock;
     * otherwise this loader and thread are added to the waitingList
     */
    private synchronized static boolean tryLock(Thread currentThread,
Object loader) {
        if (lockThread == currentThread) {
            lockCount++;
            return true;
        }
        if (lockThread == null) {
            lockCount++;
            lockThread = currentThread;
            return true;
        }
        waitingList.add(new Object[] {currentThread, loader});
        return false;
    }

    /*
     * returns true if this thread already has the global lock
     */
    private synchronized static boolean tryLock(Thread currentThread) {
        if (lockThread == currentThread) {
```

```
            lockCount++;
            return true;
        }
        return false;
    }

    /*
     * unlocks the global lock and notifies the first waiting thread that
they
     * now have the lock
     */
    private static void unlock() {
        Thread waitingThread = null;
        Object loader = null;
        synchronized (BundleLoader.class) {
            lockCount--;
            if (lockCount != 0)
                return;

            if (waitingList.isEmpty()) {
                lockThread = null;
                return;
            }

            Object[] waiting = (Object[]) waitingList.get(0);
            waitingThread = (Thread) waiting[0];
            loader = waiting[1];
        }
        synchronized (loader) {
            synchronized (BundleLoader.class) {
```

```
                lockThread = waitingThread;
                waitingList.remove(0);
                loader.notifyAll();
            }
        }
    }

}
```