Bundle Resolving Model

0 尤埃与产品简介

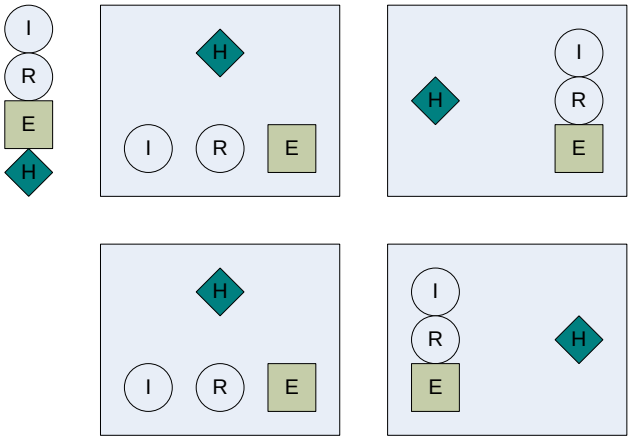
西安尤埃信息技术有限公司（http://www.uishell.com）成立于 2008 年 5 月份，专注于尤埃开放服务平台和尤埃 SaaS 引擎云计算产品开发。

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。

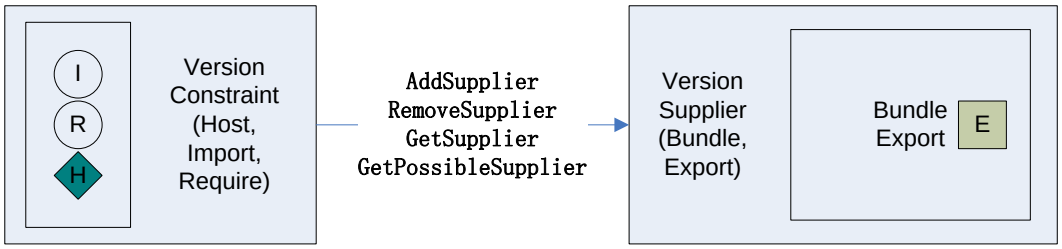
尤埃 SaaS 引擎（XAUI SaaS Engine，XSE）是一个 SaaS 应用商店**开放平台**。该平台是面向 SaaS 运营商、SaaS 提供商和 SaaS 消费者三个角色的 **PaaS 云计算平台**，其模式为“SaaS 运营商负责平台运营，SaaS 提供商利用平台提供的开发工具包基于 VS2008SP1 开发 SaaS 应用并上传，SaaS 消费者在应用商店挑选、购买并使用 SaaS 应用”。该平台由应用商店网站、应用开发工具包和应用虚拟运行环境构成。

1 Resolving Model Example

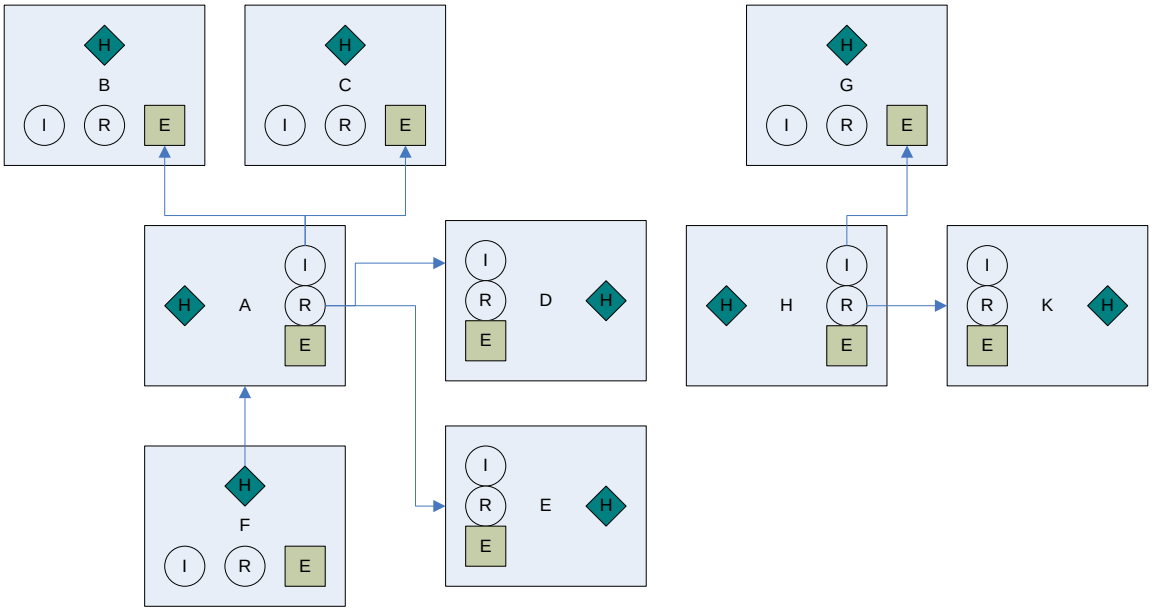
A) Icons



B) Wiring Model



C) State



OSGi 提供对系统状态的持久化。

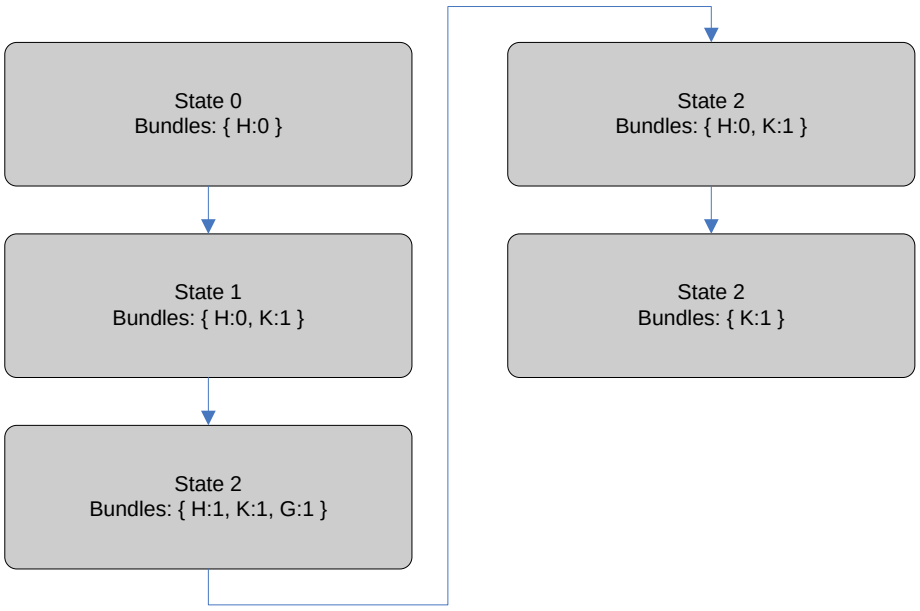
2 Resolving Process in a simple scenario

1） 安装 Bundle H，系统状态 Resolved=false，当进行解析后，所有 Bundle 的状态为{ H:0 }，因为 H 的 Import 和 Require 不能被满足，此时系统状态进入 Resolved=true。

2） 安装 Bundle K，系统状态 Resolved=false，当进行解析后，所有 Bundle 的状态为{ H:0, K:1 }，因为 H 的 Import 不能被满足，但 K 能够被解析，此时系统状态进入 Resolved=true。

3）同 2）。

4） 卸载了 Bundle G…。



对于整个系统状态，Bundle 的安装、卸载、解析和反解析将引起其它 Bundle 状态的变化，因此也将影响到系统状态。

3 Constraint and resolver layer codes

3.1 osgi.eclipse.osgi.service.resolver

该层定义的了约束和解析描述、系统状态与解析器接口。

3.1.1 BaseDescription

表示 Bundle 一个不需要被解析描述信息，比如导出描述和 Bundle 描述。

```
package org.eclipse.osgi.service.resolver;

import org.osgi.framework.Version;

public interface BaseDescription {
	//名称。
	public String getName();
	//版本。
```

```
        public Version getVersion();
        //该Description拥有Bundle。
        public BundleDescription getSupplier();
}
```

3.1.2 ExportPackageDescription

表示一个 Export 描述。

```
package org.eclipse.osgi.service.resolver;

import java.util.Map;

public interface ExportPackageDescription extends BaseDescription {
        //标识是否Re-export。
        public boolean isRoot();
        public Map getAttributes();
        public Map getDirectives();
        public Object getDirective(String key);
        //导出的Bundle。
        public BundleDescription getExporter();
}
```

3.1.3 BundleDescription

表示一个 Bundle 的描述信息，包括：约束和描述。一个 Bundle 的约束有：Host 约束、Require 约束、Import 约束等；描述有：Export 描述和 Bundle 描述。

```
package org.eclipse.osgi.service.resolver;

public interface BundleDescription extends BaseDescription {

        public String getSymbolicName();
        public String getLocation();

        //所有Require约束规范。
        public BundleSpecification[] getRequiredBundles();
```

```
//所有Export描述。
public ExportPackageDescription[] getExportPackages();
//所有Import约束规范。
public ImportPackageSpecification[] getImportPackages();

public GenericSpecification[] getGenericRequires();
public GenericDescription[] getGenericCapabilities();
//是否有动态Import。
public boolean hasDynamicImports();
//返回这个Bundle被解析器采用的Export，没有解析或没有共享包的话，返回空。
public ExportPackageDescription[] getSelectedExports();
//被解析的Require描述。
public BundleDescription[] getResolvedRequires();
//被解析的Import描述。
public ExportPackageDescription[] getResolvedImports();
//是否已经解析。
public boolean isResolved();
//保存该Bundle的State。
public State getContainingState();

public String toString();
//解析的Host。
public HostSpecification getHost();

public long getBundleId();
//所有片段描述。
public BundleDescription[] getFragments();

public boolean isSingleton();
//是否正在删除。
public boolean isRemovalPending();
//所有依赖的Bundle。
public BundleDescription[] getDependents();

public Object getUserObject();
```

```
        public void setUserObject(Object userObject);

        public String getPlatformFilter();
        //是否已经附加了Fragments。
        public boolean attachFragments();
        //是否允许动态附加片段。
        public boolean dynamicFragments();
        //要求的执行环境。
        public String[] getExecutionEnvironments();
}
```

3.1.4 VersionConstraint

表示一个约束的描述信息。

```
package org.eclipse.osgi.service.resolver;

public interface VersionConstraint extends Cloneable {
        //约束的名称。比如，Require约束的名称就是所需要的Bundle特征名。
        public String getName();
        //约束指定的版本范围。
        public VersionRange getVersionRange();
        //声明该约束的Bundle。
        public BundleDescription getBundle();
        //是否解析。
        public boolean isResolved();
        //判断一个描述是否满足该约束。
        public boolean isSatisfiedBy(BaseDescription supplier);
        //返回满足约束的描述。
        public BaseDescription getSupplier();
}
```

3.1.5 BundleSpecification

表示一个 Require-Bundle 约束规格。

```
package org.eclipse.osgi.service.resolver;
```

```
public interface BundleSpecification extends VersionConstraint {
	public boolean isExported();
	//是否为Optional。
	public boolean isOptional();
}
```

3.1.6 HostSpecification

表示一个 Host 约束规格。

```
package org.eclipse.osgi.service.resolver;

public interface HostSpecification extends VersionConstraint {
	//满足约束的Hosts。
	public BundleDescription[] getHosts();
	//是否有多个Host。
	public boolean isMultiHost();
}
```

3.1.7 ImportPackageSpecification

表示一个 Import 约束规格。

```
package org.eclipse.osgi.service.resolver;

import java.util.Map;

public interface ImportPackageSpecification extends VersionConstraint {
	public static final String RESOLUTION_STATIC = "static";
	public static final String RESOLUTION_OPTIONAL = "optional";
	public static final String RESOLUTION_DYNAMIC  = "dynamic";
	//Import的Bundle的特征名。
	public String getBundleSymbolicName();
	//Import的Bundle版本范围。
	public VersionRange getBundleVersionRange();

	public Map getAttributes();
	public Map getDirectives();
```

```
        public Object getDirective(String key);
}
```

3.1.8 ResolverError

```
//解析过程发生的错误。
package org.eclipse.osgi.service.resolver;

public interface ResolverError {
        //缺少Import、Require或片段。
        public static final int MISSING_IMPORT_PACKAGE = 0x0001;
        public static final int MISSING_REQUIRE_BUNDLE = 0x0002;
        public static final int MISSING_FRAGMENT_HOST = 0x0004;
        //单件错误。
        public static final int SINGLETON_SELECTION = 0x0008;
        //片段冲突。
        public static final int FRAGMENT_CONFLICT = 0x0010;
        //Use Import冲突。
        public static final int IMPORT_PACKAGE_USES_CONFLICT = 0x0020;
        //Require-Bundle冲突。
        public static final int REQUIRE_BUNDLE_USES_CONFLICT = 0x0040;
        //Import权限错误。
        public static final int IMPORT_PACKAGE_PERMISSION = 0x0080;
        //Export权限错误。
        public static final int EXPORT_PACKAGE_PERMISSION = 0x0100;
        //Require权限错误。
        public static final int REQUIRE_BUNDLE_PERMISSION = 0x0200;
        //Bundle提供商权限错误。
        public static final int PROVIDE_BUNDLE_PERMISSION = 0x0400;
        //Host-Bundle权限错误。
        public static final int HOST_BUNDLE_PERMISSION = 0x0800;
        //片段权限错误。
        public static final int FRAGMENT_BUNDLE_PERMISSION = 0x1000;
        //平台过滤。
        public static final int PLATFORM_FILTER = 0x2000;
```

```
        //执行环境不符。
        public static final int MISSING_EXECUTION_ENVIRONMENT = 0x4000;

        public static final int MISSING_GENERIC_CAPABILITY = 0x8000;

        //引发解析错误的Bundle。
        public BundleDescription getBundle();
        //错误的类型。
        public int getType();
        //关联的数据信息。
        public String getData();
        //没有被满足的约束。
        public VersionConstraint getUnsatisfiedConstraint();
}
```

3.1.9 Resolver

OSGi 的 Bundle 约束解析器。

```
package org.eclipse.osgi.service.resolver;

import java.util.Comparator;
import java.util.Dictionary;

public interface Resolver {

        //解析系统状态并返回在Changes描述的BundleDelta。这个方法尽在用户调用
        //State.resolve方法后调用，若Resolve方法没有被调用，State不会更新解析池。
        public void resolve(BundleDescription[] discard,
                Dictionary[] platformProperties);
        //清空所有的缓存数据。当更改Resolver的State对象改变了，进行次操作。
        public void flush();
        //返回关联的系统状态。
        public State getState();
        //设置关联的状态。只能设置一次。
        public void setState(State value);
        //通知解析器，一个Bundle添加到系统状态了。
```

```
        public void bundleAdded(BundleDescription bundle);
        //通知解析器，一个Bundle从系统状态删除。
        public void bundleRemoved(BundleDescription bundle,
                boolean pending);

        public void bundleUpdated(BundleDescription newDescription,
                BundleDescription existingDescription, boolean pending);
        //解析一个动态Import。
        public ExportPackageDescription resolveDynamicImport(
                BundleDescription importingBundle, String requestedPackage);
        //选择策略用于排序满足约束的描述。
        public void setSelectionPolicy(Comparator selectionPolicy);
        public Comparator getSelectionPolicy();
}
```

3.1.10 StateDelta

表示系统状态的变化。

```
package org.eclipse.osgi.service.resolver;

//State变更包含了在一个系统状态下所有Bundle变更。
public interface StateDelta {
        //所有Bundle变更。
        public BundleDelta[] getChanges();
        //查找变更。
        public BundleDelta[] getChanges(int mask, boolean exact);
        //关联的状态。
        public State getState();
}
```

3.1.11 BundleDelta

表示一个 Bundle 状态变化。

```
package org.eclipse.osgi.service.resolver;
```

```
public interface BundleDelta extends Comparable {
	//添加到系统状态、从系统状态删除、更新、解析、反解析、正在删除、删除完成。
	public static final int ADDED = 0x1;
	public static final int REMOVED = 0x2;
	public static final int UPDATED = 0x4;
	public static final int RESOLVED = 0x8;
	public static final int UNRESOLVED = 0x10;
	public static final int REMOVAL_PENDING = 0x80;
	public static final int REMOVAL_COMPLETE = 0x100;

	//关联的Bundle。
	public BundleDescription getBundle();
	//变更类型。
	public int getType();
	//比较。
	public int compareTo(Object obj);
}
```

3.1.12 State

表示报告给解析器的系统状态。它包括所有的 Bundle——解析的和未解析的。

```
package org.eclipse.osgi.service.resolver;

import java.util.Dictionary;

import org.osgi.framework.BundleException;
import org.osgi.framework.Version;

public interface State {
	//添加一个Bundle到系统状态。
	public boolean addBundle(BundleDescription description);
	//与一个状态相比的变化。
	public StateDelta compare(State baseState) throws BundleException;
	//从系统状态中删除一个Bundle。
	public BundleDescription removeBundle(long bundleId);
	public boolean removeBundle(BundleDescription bundle);
```

```
//更新一个Bundle。
public boolean updateBundle(BundleDescription newDescription);
//返回从上次解析时间到现在发生的变化。
public StateDelta getChanges();
//返回系统状态所有Bundle。
public BundleDescription[] getBundles();
//返回指定ID的Bundle
public BundleDescription getBundle(long id);
public BundleDescription getBundle(String symbolicName,
      Version version);
public BundleDescription getBundleByLocation(String location);
//时间戳。
public long getTimeStamp();
public void setTimeStamp(long newTimeStamp);
//如果和上次调用Resolve()方法开始到现在，系统状态没有任何变更，则返回true。
public boolean isResolved();
//使用提供的描述解析给定的约束。如果Supplier为null，则将反解析这个约束。
public void resolveConstraint(VersionConstraint constraint,
      BaseDescription supplier);
//解析一个Bundle，用于决定哪些Supplier满足哪些约束。
public void resolveBundle(BundleDescription bundle,
      boolean status, BundleDescription[] hosts,
      ExportPackageDescription[] selectedExports,
      BundleDescription[] resolvedRequires,
      ExportPackageDescription[] resolvedImports);
//删除一个Bundle。
public void removeBundleComplete(BundleDescription bundle);
//添加一个错误的系统状态。
public void addResolverError(BundleDescription bundle, int type,
      String data, VersionConstraint unsatisfied);
//删除一个Bundle关联的所有错误。
public void removeResolverErrors(BundleDescription bundle);
//返回一个Bundle的所有错误。
public ResolverError[] getResolverErrors(
      BundleDescription bundle);
//关联的解析器。
```

```
    public Resolver getResolver();
    public void setResolver(Resolver value);
    //解析当前系统状态。
    public StateDelta resolve(boolean incremental);
    public StateDelta resolve();
    public StateDelta resolve(BundleDescription[] discard);

    public void setOverrides(Object value);

    public BundleDescription[] getResolvedBundles();

    public boolean isEmpty();

    //返回系统状态所有Export描述。
    public ExportPackageDescription[] getExportedPackages();
    //返回系统状态所有Bundle。
    public BundleDescription[] getBundles(String symbolicName);

    public StateObjectFactory getFactory();
    //查找一个Import。
    public ExportPackageDescription linkDynamicImport(
          BundleDescription importingBundle, String requestedPackage);

    public boolean setPlatformProperties(
          Dictionary platformProperties);
    public boolean setPlatformProperties(
          Dictionary[] platformProperties);
    public Dictionary[] getPlatformProperties();


    public ExportPackageDescription[] getSystemPackages();
    //关联的StateHelper。
    public StateHelper getStateHelper();
    //最大的BundleID。
    public long getHighestBundleId();
}
```

3.1.13 StateHelper

```
package org.eclipse.osgi.service.resolver;


public interface StateHelper {
	public static int ACCESS_ENCOURAGED = 0x01;
	public static int ACCESS_DISCOURAGED = 0x02;
	public static int VISIBLE_INCLUDE_EE_PACKAGES = 0x01;

	//获取依赖Bundle。
	public BundleDescription[] getDependentBundles(
		BundleDescription[] bundles);
	//必要的Bundle。
	public BundleDescription[] getPrerequisites(
		BundleDescription[] bundles);
	//返回一个Bundel未满足的约束。
	public VersionConstraint[] getUnsatisfiedConstraints(
		BundleDescription bundle);
	public VersionConstraint[] getUnsatisfiedLeaves(
		BundleDescription[] bundles);
	//判断一个Import是否可以解析。
	public boolean isResolvable(
		ImportPackageSpecification specification);
	public boolean isResolvable(BundleSpecification specification);
	public boolean isResolvable(HostSpecification specification);

	public Object[][] sortBundles(BundleDescription[] toSort);
	//所有的Export。
	public ExportPackageDescription[] getVisiblePackages(
		BundleDescription bundle);
	public ExportPackageDescription[] getVisiblePackages(
		BundleDescription bundle, int options);

	public int getAccessCode(
		BundleDescription bundle, ExportPackageDescription export);
}
```

3.1.14 StateObjectFactory

```
package org.eclipse.osgi.service.resolver;

import java.io.*;
import java.util.Dictionary;
import java.util.Map;
import org.eclipse.osgi.internal.resolver.StateObjectFactoryImpl;
import org.osgi.framework.*;

public interface StateObjectFactory {

	public static final StateObjectFactory defaultFactory =
		new StateObjectFactoryImpl();

	//创建一个State。
	public State createState(boolean resolver);
	public State createState(State state);
	//创建Bundle描述。
	public BundleDescription createBundleDescription(
		long id, String symbolicName, Version version,
		String location, BundleSpecification[] required,
		HostSpecification host,
		ImportPackageSpecification[] imports,
		ExportPackageDescription[] exports, boolean singleton,
		boolean attachFragments, boolean dynamicFragments,
		String platformFilter, String[] executionEnvironments,
		GenericSpecification[] genericRequires,
		GenericDescription[] genericCapabilities);
	public BundleDescription createBundleDescription(
		State state, Dictionary manifest, String location, long id)
		throws BundleException;
	public BundleDescription createBundleDescription(
		BundleDescription original);
	//创建Bundle约束规格。
	public BundleSpecification createBundleSpecification(
		String requiredSymbolicName,
```

```
        VersionRange requiredVersionRange, boolean export,
        boolean optional);
public BundleSpecification createBundleSpecification(
        BundleSpecification original);
//创建Host约束规格。
public HostSpecification createHostSpecification(
        String hostSymbolicName, VersionRange hostVersionRange);
public HostSpecification createHostSpecification(
        HostSpecification original);
//创建Import约束规格。
public ImportPackageSpecification
createImportPackageSpecification(String packageName,
        VersionRange versionRange, String bundleSymbolicName,
        VersionRange bundleVersionRange, Map directives,
        Map attributes, BundleDescription importer);
public ImportPackageSpecification
createImportPackageSpecification(
        ImportPackageSpecification original);
//创建Export描述。
public ExportPackageDescription createExportPackageDescription(
        String packageName, Version version, Map directives,
        Map attributes, boolean root, BundleDescription exporter);
public ExportPackageDescription createExportPackageDescription(
        ExportPackageDescription original);
//创建Generic描述。
public GenericDescription createGenericDescription(String name,
        String type, Version version, Map attributes);
public GenericSpecification createGenericSpecification(
        String name, String type, String matchingFilter,
        boolean optional, boolean multiple)
        throws InvalidSyntaxException;
//持久化给定状态到一个目录。
public void writeState(State state, File stateDirectory)
        throws IOException;
//从一个给定目录读取持久化信息。
public State readState(File stateDirectory) throws IOException;
```

```
}
```

3.1.15 PlatformAdmin

系统框架服务，允许 Bundle 编程人员检查框架所知的 Bundle 和包。

```
package org.eclipse.osgi.service.resolver;

import org.osgi.framework.BundleException;

/**
 * Framework service which allows bundle programmers to inspect the
bundles and
 * packages known to the Framework.  The PlatformAdmin service also
allows bundles
 * with sufficient privileges to update the state of the framework by
committing a new
 * configuration of bundles and packages.
 *
 * If present, there will only be a single instance of this service
 * registered with the Framework.
 * <p>
 * Clients may implement this interface.
 * </p>
 * @since 3.1
 */
public interface PlatformAdmin {
	//返回一个当前系统易变的State。
	public State getState();
	//返回一个表示当前系统的状态，指定是否易变。
	public State getState(boolean mutable);
	//返回StateHelper类。
	public StateHelper getStateHelper();
	//提交当前状态与给定状态发生的变化。
	public void commit(State state) throws BundleException;
	//返回系统提供的解析器。
	public Resolver getResolver();
```

```
	//返回StateObjectFactory实例。
	public StateObjectFactory getFactory();
}
```

3.2 osgi.eclipse.osgi.internal.module

3.2.1 ResolverConstraint

和约束关联的类，表示一个连线的起点。

```
package org.eclipse.osgi.internal.module;

import org.eclipse.osgi.service.resolver.BundleDescription;
import org.eclipse.osgi.service.resolver.VersionConstraint;

public abstract class ResolverConstraint {
	final protected ResolverBundle bundle;
	final protected VersionConstraint constraint;
	private VersionSupplier[] possibleSuppliers;
	private int selectedSupplierIndex = 0;

	ResolverConstraint(ResolverBundle bundle,
		VersionConstraint constraint) {
		this.bundle = bundle;
		this.constraint = constraint;
	}
	//返回要求ResolverContraint约束的Bundle。
	// returns the Resolver bundle requiring the ResolverConstraint
	ResolverBundle getBundle() {
		return bundle;
	}
	//返回要求ResolverContainer的Description。
	BundleDescription getBundleDescription() {
		return bundle.getBundle();
	}
	//这个约束是否来自于一个片段。
	boolean isFromFragment() {
```

```
        return constraint.getBundle().getHost() != null;
}
//与VersionContraint相同，但检查的权限。
boolean isSatisfiedBy(VersionSupplier vs) {
        if (!bundle.getResolver().
                getPermissionChecker().checkPermission(constraint,
                vs.getBaseDescription()))
                return false;
        return constraint.isSatisfiedBy(vs.getBaseDescription());
}
//返回约束。
VersionConstraint getVersionConstraint() {
        return constraint;
}
//返回约束的名称。
public String getName() {
        return constraint.getName();
}

public String toString() {
        return constraint.toString();
}
//这个约束是否可选。
abstract boolean isOptional();
//设置候选Supplier。
public void setPossibleSuppliers(
        VersionSupplier[] possibleSuppliers) {
        this.possibleSuppliers = possibleSuppliers;
}
//添加候选Supplier。
void addPossibleSupplier(VersionSupplier supplier) {
        if (supplier == null)
                return;
        //由于多个Supplier比较少，因此只用一个数组。
        if (possibleSuppliers == null) {
                possibleSuppliers = new VersionSupplier[] {supplier};
                return;
```

```
        }
        VersionSupplier[] newSuppliers =
                new VersionSupplier[possibleSuppliers.length + 1];
        System.arraycopy(possibleSuppliers, 0,
                newSuppliers, 0, possibleSuppliers.length);
        newSuppliers[possibleSuppliers.length] = supplier;
        possibleSuppliers = newSuppliers;
    }

    public void removePossibleSupplier(VersionSupplier supplier) {
        if (possibleSuppliers == null || supplier == null)
                return;
        int index = -1;
        for (int i = 0; i < possibleSuppliers.length; i++) {
                if (possibleSuppliers[i] == supplier) {
                        index = i;
                        break;
                }
        }
        if (index >= 0) {
                if (possibleSuppliers.length == 1) {
                        possibleSuppliers = null;
                        return;
                }
                VersionSupplier[] newSuppliers = new VersionSupplier[
                        possibleSuppliers.length - 1];
                System.arraycopy(possibleSuppliers, 0,
                        newSuppliers, 0, index);
                if (index < possibleSuppliers.length - 1)
                        System.arraycopy(possibleSuppliers, index + 1,
                        newSuppliers, index,
                        possibleSuppliers.length - index - 1);
                possibleSuppliers = newSuppliers;
        }
    }
    //候选Supplier长度。
    int getNumPossibleSuppliers() {
        if (possibleSuppliers == null)
```

```
                return 0;
        return possibleSuppliers.length;
    }
    //选择下一个Supplier。
    boolean selectNextSupplier() {
        if (possibleSuppliers == null ||
                selectedSupplierIndex >= possibleSuppliers.length)
                return false;
        selectedSupplierIndex += 1;
        return selectedSupplierIndex < possibleSuppliers.length;
    }
    //获取相应的Supplier。
    VersionSupplier getSelectedSupplier() {
        if (possibleSuppliers == null ||
                selectedSupplierIndex >= possibleSuppliers.length)
                return null;
        return possibleSuppliers[selectedSupplierIndex];
    }
    //设置Supplier的索引。
    void setSelectedSupplier(int selectedSupplier) {
        this.selectedSupplierIndex = selectedSupplier;
    }

    int getSelectedSupplierIndex() {
        return this.selectedSupplierIndex;
    }
    //返回所有的Supplier。
    VersionSupplier[] getPossibleSuppliers() {
        return possibleSuppliers;
    }
    //清空所有的Supplier。
    void clearPossibleSuppliers() {
        possibleSuppliers = null;
        selectedSupplierIndex = 0;
    }
}
```

3.2.2 BundleConstraint

与 Require-Bundle 和 Host-Bundle 约束相关的类。连接线的起点。

```
package org.eclipse.osgi.internal.module;

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

public class BundleConstraint extends ResolverConstraint {
	BundleConstraint(ResolverBundle bundle, VersionConstraint
bundleConstraint) {
		super(bundle, bundleConstraint);
	}

	boolean isOptional() {
		if (constraint instanceof HostSpecification)
			return false;
		return ((BundleSpecification) constraint).isOptional();
	}
}
```

3.2.3 ResolverImport

与 Import 约束关联的类，表示一个 Import 约束连接线的起点。

```
package org.eclipse.osgi.internal.module;

import org.eclipse.osgi.service.resolver.ImportPackageSpecification;
import org.osgi.framework.Constants;

public class ResolverImport extends ResolverConstraint {
	//仅由动态Import使用。
	private String name;
	//关联的Bundle和约束。
	ResolverImport(ResolverBundle bundle,
		ImportPackageSpecification ips) {
		super(bundle, ips);
	}
```

```
	//是否可选Import。
	boolean isOptional() {
		return ImportPackageSpecification.RESOLUTION_OPTIONAL.
			equals(((ImportPackageSpecification) constraint).
			getDirective(Constants.RESOLUTION_DIRECTIVE));
	}
	//是否动态Import。
	boolean isDynamic() {
		return ImportPackageSpecification.RESOLUTION_DYNAMIC.
			equals(((ImportPackageSpecification) constraint).
			getDirective(Constants.RESOLUTION_DIRECTIVE));
	}
	//返回动态引用的包名。
	public String getName() {
		if (name != null)
			return name;
		return super.getName();
	}
	//设置动态Import的包名。
	void setName(String requestedPackage) {
		this.name = requestedPackage;
	}
}
```

3.2.4 VersionSupplier

一个 VersionSupplier 类用于匹配一个 BaseDescription（Bundle 和 Export）。它表示一个连接线的终端。

```
package org.eclipse.osgi.internal.module;

import org.eclipse.osgi.service.resolver.BaseDescription;
import org.eclipse.osgi.service.resolver.BundleDescription;
import org.osgi.framework.Version;

//BaseDescription用于向Contraint提供解析源，VersionSupplier表示一个
BaseDescription对应的Supplier。
```

```
public abstract class VersionSupplier {
	BaseDescription base;
	boolean dropped = false;
	//被匹配的BaseDescription。
	VersionSupplier(BaseDescription base) {
		this.base = base;
	}

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

	public String getName() {
		return base.getName();
	}

	public BaseDescription getBaseDescription() {
		return base;
	}
	//如果连接线被删除时，返回true。
	boolean isDropped() {
		return dropped;
	}
	//当提供者删除后设置连接线是否被删除。仅由VersionHashMap使用。
	void setDropped(boolean dropped) {
		this.dropped = dropped;
	}
	//返回提供VersionSupplier的Bundle。
	abstract public BundleDescription getBundle();

	public String toString() {
		return base.toString();
	}
}
```

3.2.5 ResolverExport

一个 Import 约束表示连线 Export 终端。

```
package org.eclipse.osgi.internal.module;

import org.eclipse.osgi.service.resolver.BundleDescription;
import org.eclipse.osgi.service.resolver.ExportPackageDescription;
import org.osgi.framework.Constants;
//对应一个Export的Supplier。一个Export的Supplier由ResolverBundle和
ExportNamespaceDescription组成。
public class ResolverExport extends VersionSupplier {
	private ResolverBundle resolverBundle;
	//所在Bundle和Export描述。
	ResolverExport(ResolverBundle resolverBundle,
		ExportPackageDescription epd) {
		super(epd);
		this.resolverBundle = resolverBundle;
	}

	public ExportPackageDescription getExportPackageDescription() {
		return (ExportPackageDescription) base;
	}

	public BundleDescription getBundle() {
		return getExportPackageDescription().getExporter();
	}

	ResolverBundle getExporter() {
		return resolverBundle;
	}

	String[] getUsesDirective() {
		return (String[]) getExportPackageDescription().
		getDirective(Constants.USES_DIRECTIVE);
	}
}
```

3.2.6 ResolverBundle

一个 Require 或 Host 约束的连接线的终端。

```
package org.eclipse.osgi.internal.module;

import java.util.*;
import org.eclipse.osgi.internal.resolver.ExportPackageDescriptionImpl;
import org.eclipse.osgi.service.resolver.*;
import org.osgi.framework.Constants;

//与BundleDescription匹配的对象。
public class ResolverBundle extends VersionSupplier implements
Comparable{
	public static final int UNRESOLVED = 0;
	public static final int RESOLVING = 1;
	public static final int RESOLVED = 2;

	private Long bundleID;
	private BundleConstraint host;
	private ResolverImport[] imports;
	private ResolverExport[] exports;
	private BundleConstraint[] requires;
	private GenericCapability[] capabilities;
	private GenericConstraint[] genericReqiures;
	//片段支持。
	private ArrayList fragments;
	private HashMap fragmentExports;
	private HashMap fragmentImports;
	private HashMap fragmentRequires;
	private HashMap fragmentGenericRequires;
	//指定该Bundle是否可解析。
	private boolean resolvable = true;
	//这个Bundle的内部解析状态。
	private int state = UNRESOLVED;
	//解析器。
	private ResolverImpl resolver;
```

```
private boolean newFragmentExports;
private ArrayList refs;

ResolverBundle(BundleDescription bundle, ResolverImpl resolver) {
	super(bundle);
	this.bundleID = new Long(bundle.getBundleId());
	this.resolver = resolver;
	initialize(bundle.isResolved());
}
//初始化Host约束、Require-Bundle约束、Import约束和Export描述。
void initialize(boolean useSelectedExports) {
	if (getBundle().isSingleton())
		refs = new ArrayList();
	//一般约束。
	GenericDescription[] actualCapabilities =
		getBundle().getGenericCapabilities();
	capabilities =
		new GenericCapability[actualCapabilities.length];
	for (int i = 0; i < capabilities.length; i++)
		capabilities[i] = new GenericCapability(
		this, actualCapabilities[i]);
	//如果该Bundle是一个片段，则建立Host约束，然后返回。
	if (getBundle().getHost() != null) {
		host = new BundleConstraint(
			this, getBundle().getHost());//建立一个Host约束。
		exports = new ResolverExport[0];
		imports = new ResolverImport[0];
		requires = new BundleConstraint[0];
		genericReqiures = new GenericConstraint[0];
		return;
	}

	ImportPackageSpecification[] actualImports =
		getBundle().getImportPackages();
	//重新排列Import，将可选Import放到最后，建立Import约束。
	ArrayList importList = new ArrayList(actualImports.length);
	for (int i = actualImports.length - 1; i >= 0; i--)
```

```
            if (ImportPackageSpecification.RESOLUTION_OPTIONAL.
                  equals(actualImports[i].getDirective(
                  Constants.RESOLUTION_DIRECTIVE)))
                  importList.add(new ResolverImport(
                  this, actualImports[i]));
            else
                  importList.add(0, new ResolverImport(
                  this, actualImports[i]));
      imports = (ResolverImport[]) importList.toArray(
            new ResolverImport[importList.size()]);
      //建立所有的Export描述，包括UseExports。
      ExportPackageDescription[] actualExports =
            useSelectedExports ? getBundle().getSelectedExports()
            : getBundle().getExportPackages();
      exports = new ResolverExport[actualExports.length];
      for (int i = 0; i < actualExports.length; i++)
            exports[i] = new ResolverExport(this,
            actualExports[i]);
      //建立所有的Require约束。
      BundleSpecification[] actualRequires =
            getBundle().getRequiredBundles();
      requires = new BundleConstraint[actualRequires.length];
      for (int i = 0; i < requires.length; i++)
            requires[i] = new BundleConstraint(this,
            actualRequires[i]);

      GenericSpecification[] actualGenericRequires =
            getBundle().getGenericRequires();
      genericReqiures = new GenericConstraint[
            actualGenericRequires.length];
      for (int i = 0; i < genericReqiures.length; i++)
            genericReqiures[i] = new GenericConstraint(
            this, actualGenericRequires[i]);

      fragments = null;
      fragmentExports = null;
      fragmentImports = null;
      fragmentRequires = null;
```

```
        fragmentGenericRequires = null;
    }
    //获取一个指定包名的Export，所有Export的第一个。
    ResolverExport getExport(String name) {
        ResolverExport[] allExports = getExports(name);
        return allExports.length == 0 ? null : allExports[0];
    }
    //获取Resolver中所有可用的Export。
    ResolverExport[] getExports(String name) {
        ArrayList results = new ArrayList(1); //一般只有1个。
        Object[] resolverExports =
        resolver.getResolverExports().get(name);
        for (int i = 0; i < resolverExports.length; i++)
            if (((ResolverExport)resolverExports[i]).
                getExporter() == this)
                results.add(resolverExports[i]);
        return (ResolverExport[]) results.toArray(
            new ResolverExport[results.size()]);
    }
    //清空连接线。
    void clearWires() {
        //清空Import。
        ResolverImport[] allImports = getImportPackages();
        for (int i = 0; i < allImports.length; i++)
            allImports[i].clearPossibleSuppliers();
        //清空Host。
        if (host != null)
            host.clearPossibleSuppliers();
        //清空Require。
        BundleConstraint[] allRequires = getRequires();
        for (int i = 0; i < allRequires.length; i++)
            allRequires[i].clearPossibleSuppliers();
        //清空一般约束的Supplier。
        GenericConstraint[] allGenericRequires =
            getGenericRequires();
        for (int i = 0; i < allGenericRequires.length; i++)
            allGenericRequires[i].setMatchingCapability(null);
```

```
}
//是否已经解析。
boolean isResolved() {
	return getState() == ResolverBundle.RESOLVED;
}
//是否片段。
boolean isFragment() {
	return host != null;
}
//返回解析状态，与Bundle状态不同。
int getState() {
	return state;
}

void setState(int state) {
	this.state = state;
}
//获取Host或片段的Import。
ResolverImport[] getImportPackages() {
	if (isFragment())//片段没有Import约束。
		return new ResolverImport[0];
	if (fragments == null || fragments.size() == 0)
		return imports;
	//如果是Host，其Import是所有片段的Import加上Host的Import。
	ArrayList resultList = new ArrayList(imports.length);
	for (int i = 0; i < imports.length; i++)
		resultList.add(imports[i]);
	for (Iterator iter = fragments.iterator(); iter.hasNext();)
	{
		ResolverBundle fragment = (ResolverBundle)
			iter.next();
		ArrayList fragImports = (ArrayList)
			fragmentImports.get(fragment.bundleID);
		if (fragImports != null)
			resultList.addAll(fragImports);
	}
	return (ResolverImport[]) resultList.toArray(
```

```
                new ResolverImport[resultList.size()]);
    }
    //获取Host或片段的Export。
    ResolverExport[] getExportPackages() {
          if (isFragment())//片段没有Export。
                return new ResolverExport[0];
          if (fragments == null || fragments.size() == 0)
                return exports;
          ArrayList resultList = new ArrayList(exports.length);
          for (int i = 0; i < exports.length; i++)
                resultList.add(exports[i]);
          for (Iterator iter = fragments.iterator(); iter.hasNext();)
          {
                ResolverBundle fragment = (ResolverBundle)
                      iter.next();
                ArrayList fragExports = (ArrayList)
                      fragmentExports.get(fragment.bundleID);
                if (fragExports != null)
                      resultList.addAll(fragExports);
          }
          return (ResolverExport[]) resultList.toArray(
                new ResolverExport[resultList.size()]);
    }
    //获取真实的Export，如果一个Bundle被解析了，则一些Export可能无效。
    //这个方法将删除无效的Export。
    ResolverExport[] getSelectedExports() {
          ResolverExport[] allExports = getExportPackages();
          int removedExports = 0;
          for (int i = 0; i < allExports.length; i++)
                if (allExports[i].isDropped())
                      removedExports++;
          if (removedExports == 0)
                return allExports;
          ResolverExport[] selectedExports = new
                ResolverExport[allExports.length - removedExports];
          int index = 0;
          for (int i = 0; i < allExports.length; i++) {
```

```
                if (allExports[i].isDropped())
                        continue;
                selectedExports[index] = allExports[i];
                index++;
        }
        return selectedExports;
}
//返回Host约束。
BundleConstraint getHost() {
        return host;
}

GenericCapability[] getGenericCapabilities() {
        return capabilities;
}
//返回Host或Fragment的Require-Bundle约束。
BundleConstraint[] getRequires() {
        if (isFragment())
                return new BundleConstraint[0];
        if (fragments == null || fragments.size() == 0)
                return requires;
        ArrayList resultList = new ArrayList(requires.length);
        for (int i = 0; i < requires.length; i++)
                resultList.add(requires[i]);
        for (Iterator iter = fragments.iterator(); iter.hasNext();)
        {
                ResolverBundle fragment = (ResolverBundle)
                        iter.next();
                ArrayList fragRequires = (ArrayList)
                        fragmentRequires.get(fragment.bundleID);
                if (fragRequires != null)
                        resultList.addAll(fragRequires);
        }
        return (BundleConstraint[]) resultList.toArray(
                new BundleConstraint[resultList.size()]);
}
//返回一般约束。
GenericConstraint[] getGenericRequires() {
```

```
        if (isFragment() || fragments == null ||
                fragments.size() == 0)
                return genericReqiures;
        ArrayList resultList = new
                ArrayList(genericReqiures.length);
        for (int i = 0; i < genericReqiures.length; i++)
                resultList.add(genericReqiures[i]);
        for (Iterator iter = fragments.iterator(); iter.hasNext();)
        {
                ResolverBundle fragment = (ResolverBundle)
                        iter.next();
                ArrayList fragGenericRegs = (ArrayList)
                        fragmentGenericRequires.get(fragment.bundleID);
                if (fragGenericRegs != null)
                        resultList.addAll(fragGenericRegs);
        }
        return (GenericConstraint[]) resultList.toArray(
                new GenericConstraint[resultList.size()]);
}
//返回一个指定名称的Require约束。
BundleConstraint getRequire(String name) {
        BundleConstraint[] allRequires = getRequires();
        for (int i = 0; i < allRequires.length; i++)
                if (allRequires[i].getVersionConstraint().
                        getName().equals(name))
                        return allRequires[i];
        return null;
}
//获取关联Bundle描述。
public BundleDescription getBundle() {
        return (BundleDescription) getBaseDescription();
}
//获取指定包名的Import。
ResolverImport getImport(String name) {
        ResolverImport[] allImports = getImportPackages();
        for (int i = 0; i < allImports.length; i++) {
                if (allImports[i].getName().equals(name)) {
                        return allImports[i];
```

```
                }
        }
        return null;
}

public String toString() {
        return "[" + getBundle() + "]";
}
//初始化片段。
private void initFragments() {
        if (fragments == null)
                fragments = new ArrayList(1);
        if (fragmentExports == null)
                fragmentExports = new HashMap(1);
        if (fragmentImports == null)
                fragmentImports = new HashMap(1);
        if (fragmentRequires == null)
                fragmentRequires = new HashMap(1);
        if (fragmentGenericRequires == null)
                fragmentGenericRequires = new HashMap(1);
}

private boolean isImported(String packageName) {
        ResolverImport[] allImports = getImportPackages();
        for (int i = 0; i < allImports.length; i++)
                if (packageName.equals(allImports[i].getName()))
                        return true;

        return false;
}

private boolean isExported(String packageName) {
        ResolverExport export = getExport(packageName);
        if (export == null)
                return false;
        return 0 > ((Integer) export.getExportPackageDescription().
                getDirective(ExportPackageDescriptionImpl.EQUINOX_EE)
                ).intValue();
```

```
}

private boolean isRequired(String bundleName) {
	return getRequire(bundleName) != null;
}
//附加一个片段。
ResolverExport[] attachFragment(
	ResolverBundle fragment, boolean dynamicAttach) {
	if (isFragment())//如果该Bundle是一个片段，则直接返回。
		return new ResolverExport[0];
	//Host不允许附加片段。
	if (!getBundle().attachFragments() ||
		(isResolved() && !getBundle().dynamicFragments()))
		return new ResolverExport[0];
	//Host不允许附加一个具有多个Host的片段。
	if (fragment.getHost().getNumPossibleSuppliers() > 0
		&& !((HostSpecification) fragment.getHost().
		getVersionConstraint()).isMultiHost())
		return new ResolverExport[0];

	ImportPackageSpecification[] newImports =
		fragment.getBundle().getImportPackages();
	BundleSpecification[] newRequires =
		fragment.getBundle().getRequiredBundles();
	ExportPackageDescription[] newExports =
		fragment.getBundle().getExportPackages();
	GenericSpecification[] newGenericRequires =
		fragment.getBundle().getGenericRequires();

	//如果是动态附加但有冲突。
	if (dynamicAttach &&
		constraintsConflict(fragment.getBundle(),
		newImports, newRequires, newGenericRequires))
		return new ResolverExport[0]; //不允许添加冲突的约束。
	//片段被解析，更新Export。
	if (isResolved() && newExports.length > 0)
		fragment.setNewFragmentExports(true);
```

```
//初始化片段。
initFragments();
if (fragments.contains(fragment))
      return new ResolverExport[0]; //如果已经包含，则返回。
fragments.add(fragment);
//解析片段。
fragment.getHost().addPossibleSupplier(this);
//附加Import。
if (newImports.length > 0) {
      ArrayList hostImports =
            new ArrayList(newImports.length);
      for (int i = 0; i < newImports.length; i++)
            if (!isImported(newImports[i].getName()))
                  hostImports.add(new ResolverImport(this,
                  newImports[i]));
      fragmentImports.put(fragment.bundleID, hostImports);
}
//附加Require。
if (newRequires.length > 0) {
      ArrayList hostRequires = new
            ArrayList(newRequires.length);
      for (int i = 0; i < newRequires.length; i++)
            if (!isRequired(newRequires[i].getName()))
                  hostRequires.add(new BundleConstraint(
                  this, newRequires[i]));
     fragmentRequires.put(fragment.bundleID,  hostRequires);
}
//附加一般约束。
if (newGenericRequires.length > 0) {
      ArrayList hostGenericRequires = new
            ArrayList(newGenericRequires.length);
      for (int i = 0; i < newGenericRequires.length; i++)
            hostGenericRequires.add(new GenericConstraint(
            this, newGenericRequires[i]));
      fragmentGenericRequires.put(fragment.bundleID,
            hostGenericRequires);
}
```

```
        //附加Export到系统状态中。
        ArrayList hostExports = new ArrayList(newExports.length);
        if (newExports.length > 0 && dynamicAttach) {
              StateObjectFactory factory =
                    resolver.getState().getFactory();
              for (int i = 0; i < newExports.length; i++) {
                    if (!isExported(newExports[i].getName())) {
                          ExportPackageDescription hostExport =
                          factory.createExportPackageDescription(
                                newExports[i].getName(),
                                newExports[i].getVersion(),
                                newExports[i].getDirectives(),
                                 newExports[i].getAttributes(),
                                newExports[i].isRoot(),
                                getBundle());
                          hostExports.add(new ResolverExport(
                                this, hostExport));
                    }
              }
              fragmentExports.put(fragment.bundleID, hostExports);
        }
        return (ResolverExport[]) hostExports.toArray(
              new ResolverExport[hostExports.size()]);
  }
  //片段冲突检测。
  boolean constraintsConflict(BundleDescription fragment,
        ImportPackageSpecification[] newImports,
        BundleSpecification[] newRequires,
        GenericSpecification[] newGenericRequires) {

        boolean result = false;
        for (int i = 0; i < newImports.length; i++) {
              ResolverImport hostImport =
                    getImport(newImports[i].getName());
              ResolverExport resolvedExport = (ResolverExport)
                    (hostImport == null ? null :
                    hostImport.getSelectedSupplier());
              //如果已解析但新Import和对应Export不匹配，则不能解析。
```

```
                if ((resolvedExport == null && isResolved()) ||
                        (resolvedExport != null
                        && !newImports[i].isSatisfiedBy(resolvedExport.
                        getExportPackageDescription()))) {
                        result = true;
                        resolver.getState().addResolverError(
                                fragment,  ResolverError.FRAGMENT_CONFLICT,
                                newImports[i].toString(), newImports[i]);
                }
        }
        //同上。
        for (int i = 0; i < newRequires.length; i++) {
                BundleConstraint hostRequire =
                        getRequire(newRequires[i].getName());
                ResolverBundle resolvedRequire = (ResolverBundle)
                        (hostRequire == null ? null :
                        hostRequire.getSelectedSupplier());
                if ((resolvedRequire == null && isResolved()) ||
                        (resolvedRequire != null
                        && !newRequires[i].isSatisfiedBy(
                        resolvedRequire.getBundle()))) {
                        result = true;
                        resolver.getState().addResolverError(
                                fragment,  ResolverError.FRAGMENT_CONFLICT,
                                newRequires[i].toString(),
                                newRequires[i]);
                }
        }
        if (isResolved() && newGenericRequires != null &&
                newGenericRequires.length > 0)
                result = true;
        return result;
}

private void setNewFragmentExports(boolean newFragmentExports) {
        this.newFragmentExports = newFragmentExports;
}
```

```
boolean isNewFragmentExports() {
	return newFragmentExports;
}
//因约束而分离片段。
ResolverExport[] detachFragment(ResolverBundle fragment,
	ResolverConstraint reason) {
	if (isFragment())
		return new ResolverExport[0];
	initFragments();
	//删除片段。
	if (!fragments.remove(fragment))
		return new ResolverExport[0];
	//分离Import、Export、Require和一般。
	fragment.setNewFragmentExports(false);
	fragment.getHost().removePossibleSupplier(this);
	ArrayList fragImports = (ArrayList)
		fragmentImports.remove(fragment.bundleID);
	ArrayList fragRequires = (ArrayList)
		fragmentRequires.remove(fragment.bundleID);
	ArrayList removedExports = (ArrayList)
		fragmentExports.remove(fragment.bundleID);
	fragmentGenericRequires.remove(fragment.bundleID);
	if (reason != null) {
		ResolverBundle[] remainingFrags =
			(ResolverBundle[]) fragments.toArray(
			new ResolverBundle[fragments.size()]);
		for (int i = 0; i < remainingFrags.length; i++) {
			resolver.getResolverExports().
			remove(detachFragment(remainingFrags[i],  null));
			VersionConstraint[] constraints;
			//什么约束不满足?
			if (reason instanceof ResolverImport)
				constraints = remainingFrags[i].
					getBundle().getImportPackages();
			else
				constraints = remainingFrags[i].
					getBundle().getRequiredBundles();
```

```
                for (int j = 0; j < constraints.length; j++)
                    if (reason.getName().equals(
                            constraints[j].getName()))
                            continue;
                resolver.getResolverExports().put(
                    attachFragment(remainingFrags[i], true));
                ArrayList newImports =
                    (ArrayList) fragmentImports.get(
                    remainingFrags[i].bundleID);
                if (newImports != null && fragImports != null)
                    for (Iterator iNewImports =
                            newImports.iterator();
                            iNewImports.hasNext();) {
                            ResolverImport newImport =
                           (ResolverImport)  iNewImports.next();
                            for (Iterator iOldImports =
                                  fragImports.iterator();
                                  iOldImports.hasNext();) {
                                  ResolverImport oldImport =
                                        (ResolverImport)
                                        iOldImports.next();
                                  if (newImport.getName().
                                  equals(oldImport.getName()))
                                        newImport.
                                        setPossibleSuppliers(
                                        oldImport.
                                        getPossibleSuppliers());
                            }
                    }
                ArrayList newRequires =
                    (ArrayList) fragmentRequires.get(
                    remainingFrags[i].bundleID);
               if (newRequires != null && fragRequires != null)
                    //同上
                            }
                    }
        }
}
```

```
		return removedExports == null ? new ResolverExport[0] :
				(ResolverExport[]) removedExports.toArray(
				new ResolverExport[removedExports.size()]);
	}

	void detachAllFragments() {
		if (fragments == null)
			return;
		ResolverBundle[] allFragments =
				(ResolverBundle[]) fragments.toArray(
					new ResolverBundle[fragments.size()]);
		for (int i = 0; i < allFragments.length; i++)
			detachFragment(allFragments[i], null);
	}

	boolean isResolvable() {
		return resolvable;
	}

	void setResolvable(boolean resolvable) {
		this.resolvable = resolvable;
	}

	void addExport(ResolverExport re) {
		ResolverExport[] newExports =
				new ResolverExport[exports.length + 1];
		for (int i = 0; i < exports.length; i++)
			newExports[i] = exports[i];
		newExports[exports.length] = re;
		exports = newExports;
	}

	ResolverImpl getResolver() {
		return resolver;
	}

	void clearRefs() {
		if (refs != null)
```

```
                refs.clear();
        }

        void addRef(ResolverBundle ref) {
                if (refs != null && !refs.contains(ref))
                        refs.add(ref);
        }

        int getRefs() {
                return refs == null ? 0 : refs.size();
        }

        ResolverBundle[] getFragments() {
                return  fragments == null ? new ResolverBundle[0] :
                        (ResolverBundle[]) fragments.toArray(
                                new ResolverBundle[fragments.size()]);
        }

        public int compareTo(Object o) {
                String bsn = getName();
                String otherBsn = ((ResolverBundle) o).getName();
                if (bsn == null)
                        return otherBsn == null ? 0 : 1;
                return otherBsn == null ? -1 : bsn.compareTo(otherBsn);
        }
}
```

3.2.7 MappedList

Key 与数组 Object[]对应的 Hash 表。

```
package org.eclipse.osgi.internal.module;

import java.util.*;

public class MappedList {
        protected HashMap internal = new HashMap();
```

```
public void put(Object key, Object value) {
	Object[] existing = (Object[]) internal.get(key);
	if (existing == null) {
		existing = new Object[1];
		existing[0] = value;
		internal.put(key, existing);
	} else {
		Object[] newValues = new Object[existing.length + 1];
		System.arraycopy(existing, 0, newValues, 0,
			existing.length);
		newValues[existing.length] = value; //对新值进行排序。
		sort(newValues);
		internal.put(key, newValues);
	}
}

protected void sort(Object[] values) {
	//自定义排序。
}

public Object[] remove(Object key) {
	return get(key, true);
}

public Object[] get(Object key) {
	return get(key, false);
}

private Object[] get(Object key, boolean remove) {
	Object[] result = (Object[]) (remove ?
		internal.remove(key) : internal.get(key));
	return result == null ? new Object[0] : result;
}
public int getSize() {
	return internal.size();
}
public Object[] getAllValues() {
	if (getSize() == 0)
```

```
                return new Object[0];
            ArrayList results = new ArrayList(getSize());
            Iterator iter = internal.values().iterator();
            while (iter.hasNext()) {
                Object[] values = (Object[]) iter.next();
                for (int i = 0; i < values.length; i++)
                    results.add(values[i]);
            }
            return results.toArray();
        }
        public void clear() {
            internal.clear();
        }

}
```

3.2.8 VersionHashMap

```
package org.eclipse.osgi.internal.module;

import java.util.*;
import org.eclipse.osgi.framework.internal.core.Constants;

public class VersionHashMap extends MappedList implements Comparator {
        private final String systemBundle =
                Constants.getInternalSymbolicName();
        private ResolverImpl resolver;

        public VersionHashMap(ResolverImpl resolver) {
            this.resolver = resolver;
        }

        protected void sort(Object[] values) {
            Arrays.sort(values, this);
        }
        //VersionSupplier.Name->VersionSupplier[]。
        public void put(VersionSupplier[] versionSuppliers) {
```

```
		for (int i = 0; i < versionSuppliers.length; i++)
			put(versionSuppliers[i].getName(),
					versionSuppliers[i]);
	}
	//设置Dropper为false。
	public void put(Object key, Object value) {
		super.put(key, value);
		((VersionSupplier) value).setDropped(false);
	}

	public boolean contains(VersionSupplier vs) {
		return contains(vs, false) != null;
	}

	private VersionSupplier contains(
		VersionSupplier vs, boolean remove) {
		Object[] existing = (Object[]) internal.get(vs.getName());
		if (existing == null)
			return null;
		for (int i = 0; i < existing.length; i++)
			if (existing[i] == vs) {
				if (remove) {
					vs.setDropped(true);
					if (existing.length == 1) {
						internal.remove(vs.getName());
						return vs;
					}
					Object[] newExisting = new
						Object[existing.length - 1];
					System.arraycopy(existing, 0,
						newExisting, 0, i);
					if (i + 1 < existing.length)
						System.arraycopy(existing, i + 1,
						newExisting, i,
						existing.length - i - 1);
					internal.put(vs.getName(), newExisting);
				}
				return vs;
```

```
            }
        return null;
    }

    public Object remove(VersionSupplier toBeRemoved) {
        return contains(toBeRemoved, true);
    }

    public void remove(VersionSupplier[] versionSuppliers) {
        for (int i = 0; i < versionSuppliers.length; i++)
            remove(versionSuppliers[i]);
    }

    public Object[] remove(Object key) {
        Object[] results = super.remove(key);
        for (int i = 0; i < results.length; i++)
            ((VersionSupplier) results[i]).setDropped(true);
        return results;
    }

    //排序。
    void reorder() {
        for (Iterator it = internal.values().iterator();
            it.hasNext();) {
            Object[] existing = (Object[]) it.next();
            if (existing.length <= 1)
                continue;
            sort(existing);
        }
    }
    //降序VersionSupplier比较器，属性是解析状态、版本和ID。
    public int compare(Object o1, Object o2) {
        if (!(o1 instanceof VersionSupplier) ||
            !(o2 instanceof VersionSupplier))
            throw new IllegalArgumentException();
        VersionSupplier vs1 = (VersionSupplier) o1;
        VersionSupplier vs2 = (VersionSupplier) o2;
```

```
            if (resolver.getSelectionPolicy() != null)
                  return resolver.getSelectionPolicy().
                        compare(vs1.getBaseDescription(),
                        vs2.getBaseDescription());
            if (systemBundle.equals(vs1.getBundle().getSymbolicName())
            && !systemBundle.equals(vs2.getBundle().getSymbolicName()))
                  return -1;
            else if (!systemBundle.equals(
                  vs1.getBundle().getSymbolicName()) &&
                  systemBundle.equals(
                  vs2.getBundle().getSymbolicName()))
                  return 1;
            if (vs1.getBundle().isResolved() !=
                  vs2.getBundle().isResolved())
                  return vs1.getBundle().isResolved() ? -1 : 1;
            int versionCompare = -(vs1.getVersion().compareTo(
                  vs2.getVersion()));
            if (versionCompare != 0)
                  return versionCompare;
            return vs1.getBundle().getBundleId() <=
                  vs2.getBundle().getBundleId() ? -1 : 1;
      }
}
```

3.2.9 ResolverImpl

```
package org.eclipse.osgi.internal.module;

import java.util.*;
import org.eclipse.osgi.framework.adaptor.FrameworkAdaptor;
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.framework.debug.FrameworkDebugOptions;
import org.eclipse.osgi.internal.module.GroupingChecker.PackageRoots;
import org.eclipse.osgi.internal.resolver.BundleDescriptionImpl;
import org.eclipse.osgi.internal.resolver.ExportPackageDescriptionImpl;
import org.eclipse.osgi.service.resolver.*;
import org.eclipse.osgi.util.ManifestElement;
import org.osgi.framework.*;
```

```
public class ResolverImpl implements
org.eclipse.osgi.service.resolver.Resolver {
	//Debug字段。
	private static final String RESOLVER =
		FrameworkAdaptor.FRAMEWORK_SYMBOLICNAME + "/resolver";
	private static final String OPTION_DEBUG = RESOLVER + "/debug";
	private static final String OPTION_WIRING = RESOLVER + "/wiring";
	private static final String OPTION_IMPORTS = RESOLVER +
		"/imports";
	private static final String OPTION_REQUIRES = RESOLVER +
		"/requires";
	private static final String OPTION_GENERICS = RESOLVER +
		"/generics";
	private static final String OPTION_GROUPING = RESOLVER +
		"/grouping";
	private static final String OPTION_CYCLES = RESOLVER + "/cycles";
	public static boolean DEBUG = false;
	public static boolean DEBUG_WIRING = false;
	public static boolean DEBUG_IMPORTS = false;
	public static boolean DEBUG_REQUIRES = false;
	public static boolean DEBUG_GENERICS = false;
	public static boolean DEBUG_GROUPING = false;
	public static boolean DEBUG_CYCLES = false;
	private static int MAX_MULTIPLE_SUPPLIERS_MERGE = 10;
	private static long MAX_COMBINATIONS = 1000000;

	private static String[][] CURRENT_EES;

	//关联的系统状态。
	private State state;
	private PermissionChecker permissionChecker;
	//暂停删除的Bundle。
	private MappedList removalPending = new MappedList();
	private boolean initialized = false;
	//Export仓库。
	private VersionHashMap resolverExports = null;
```

```
//Bundle仓库。
private VersionHashMap resolverBundles = null;
//Generics仓库。
private VersionHashMap resolverGenerics = null;
//未解析Bundle。TODO:改成集合。
private ArrayList unresolvedBundles = null;
//BundleDescription->ResolverBundle序列。
private HashMap bundleMapping = null;
private GroupingChecker groupingChecker;
private Comparator selectionPolicy;
private boolean developmentMode = false;

public ResolverImpl(
      BundleContext context, boolean checkPermissions) {
      this.permissionChecker = new PermissionChecker(
            context, checkPermissions, this);
}

PermissionChecker getPermissionChecker() {
      return permissionChecker;
}

//初始化解析器。完成：从State获取所有的BundleDescription，创建对应的
//ResolverBundle、ResolverExport，添加到仓库，并且Attach所有的片段。
private void initialize() {
      resolverExports = new VersionHashMap(this);
      resolverBundles = new VersionHashMap(this);
      resolverGenerics = new VersionHashMap(this);
      unresolvedBundles = new ArrayList();
      bundleMapping = new HashMap();
      BundleDescription[] bundles = state.getBundles();
      groupingChecker = new GroupingChecker();

      ArrayList fragmentBundles = new ArrayList();
      //添加每一个Bundle到解析器的内部状态。
      for (int i = 0; i < bundles.length; i++)
            initResolverBundle(
```

```
                bundles[i], fragmentBundles, false);
        //添加每一个正在删除的Bundle到解析器内部状态。
        Object[] removedBundles = removalPending.getAllValues();
        for (int i = 0; i < removedBundles.length; i++)
              initResolverBundle((BundleDescription)
                    removedBundles[i], fragmentBundles, true);
        //附加片段。
        for (Iterator iter = fragmentBundles.iterator();
              iter.hasNext();) {
              ResolverBundle fragment = (ResolverBundle)
                    iter.next();
              BundleDescription[] hosts = ((HostSpecification)
                    fragment.getHost().
                    getVersionConstraint()).getHosts();
              for (int i = 0; i < hosts.length; i++) {
                    ResolverBundle host = (ResolverBundle)
                          bundleMapping.get(hosts[i]);
                    if (host != null)
                          //不要添加片段Export，Host会自己做。
                          host.attachFragment(fragment, false);
              }
        }
        rewireBundles(); //重新建立连接线。
        setDebugOptions();
        initialized = true;
  }

  private void initResolverBundle(
        BundleDescription bundleDesc,
        ArrayList fragmentBundles, boolean pending) {
        ResolverBundle bundle = new ResolverBundle(
              bundleDesc, this);
        bundleMapping.put(bundleDesc, bundle);
        //直接将Export、Bundle添加到仓库。
        if (!pending || bundleDesc.isResolved()) {
              resolverExports.put(bundle.getExportPackages());
              resolverBundles.put(bundle.getName(), bundle);
```

```
                resolverGenerics.put(bundle.getGenericCapabilities());
        }
        //添加到片段中。
        if (bundleDesc.isResolved()) {
                bundle.setState(ResolverBundle.RESOLVED);
                if (bundleDesc.getHost() != null)
                        fragmentBundles.add(bundle);
        } else {
                if (!pending)
                        unresolvedBundles.add(bundle);
        }
}

//重新连线已经解析的Bundle。
private void rewireBundles() {
        ArrayList visited = new ArrayList(bundleMapping.size());
        for (Iterator iter = bundleMapping.values().iterator();
                iter.hasNext();) {
                ResolverBundle rb = (ResolverBundle) iter.next();
                if (!rb.getBundle().isResolved() || rb.isFragment())
                        continue;
                rewireBundle(rb, visited);
        }
}
//重新连线指定Bundle。
private void rewireBundle(ResolverBundle rb, ArrayList visited) {
        if (visited.contains(rb))
                return;
        visited.add(rb);
        //连线Require。
        BundleConstraint[] requires = rb.getRequires();
        for (int i = 0; i < requires.length; i++) {
                rewireRequire(requires[i], visited);
        }
        //连线Import。
        ResolverImport[] imports = rb.getImportPackages();
        for (int i = 0; i < imports.length; i++) {
```

```
                rewireImport(imports[i], visited);
        }
        //连线Generic。
        GenericConstraint[] genericRequires =
                rb.getGenericRequires();
        for (int i = 0; i < genericRequires.length; i++)
                rewireGeneric(genericRequires[i], visited);
}

private void rewireGeneric(GenericConstraint constraint,
        ArrayList visited) {
        if (constraint.getMatchingCapabilities() != null)
                return;
        GenericDescription[] suppliers = ((GenericSpecification)
                constraint.getVersionConstraint()).getSuppliers();
        if (suppliers == null)
                return;
        //获取名称匹配的Des。
        Object[] matches = resolverGenerics.get(
                constraint.getName());
        for (int i = 0; i < matches.length; i++) {
                GenericCapability match =
                        (GenericCapability) matches[i];
                for (int j = 0; j < suppliers.length; j++)
                        if (match.getBaseDescription() == suppliers[j])
                                constraint.setMatchingCapability(match);
        }
        GenericCapability[] matchingCapabilities =
                constraint.getMatchingCapabilities();
        if (matchingCapabilities != null)
                for (int i = 0; i < matchingCapabilities.length; i++)
                        rewireBundle(
                        matchingCapabilities[i].getResolverBundle(),
                        visited);
}
//连接RequireBundles，除了连接到Require-Bundle，还要求将Required-
//Bundle也重新连线。
```

```
private void rewireRequire(
        BundleConstraint req, ArrayList visited) {
        if (req.getSelectedSupplier() != null) //还未解析过。
                return;
        //获取约束匹配的Bundle。
        ResolverBundle matchingBundle =
                (ResolverBundle) bundleMapping.get(
                req.getVersionConstraint().getSupplier());
        req.addPossibleSupplier(matchingBundle); //连线。
        if (matchingBundle == null && !req.isOptional()) {
                System.err.println("Could not find matching bundle
                        for " + req.getVersionConstraint());
        }
        if (matchingBundle != null) { //强制连线Require的Bundle。
                rewireBundle(matchingBundle, visited);
        }
}
//连线Import，并对Export-Bundle进行重新画线。
private void rewireImport(ResolverImport imp, ArrayList visited)
{
        //动态和未解析的Import不能重新画线。
        if (imp.isDynamic() || imp.getSelectedSupplier() != null)
                return;

        ResolverExport matchingExport = null;
        ExportPackageDescription importSupplier =
                (ExportPackageDescription) imp.
                getVersionConstraint().getSupplier();
        ResolverBundle exporter = importSupplier == null ? null :
                (ResolverBundle) bundleMapping.get(
                        importSupplier.getExporter());
        //获取匹配的Export。
        Object[] matches = resolverExports.get(imp.getName());
        for (int j = 0; j < matches.length; j++) {
                ResolverExport export = (ResolverExport) matches[j];
                if (export.getExporter() == exporter &&
                        importSupplier == export.
```

```
                        getExportPackageDescription()) {
                        matchingExport = export;
                        break;
                }
        }
        imp.addPossibleSupplier(matchingExport);
        //检查该Export是否为Use导出。
        if (matchingExport == null && exporter != null) {
                ResolverExport reprovidedExport =
                        new ResolverExport(exporter, importSupplier);
                if (exporter.getExport(imp.getName()) == null) {
                        exporter.addExport(reprovidedExport);
                        resolverExports.put(
                                reprovidedExport.getName(),
                                reprovidedExport);
                }
                imp.addPossibleSupplier(reprovidedExport);
        }
        //如果Supplier为空且不是Optional-Import，则错误。
        if (imp.getSelectedSupplier() == null && !imp.isOptional())
        {
                System.err.println("Could not find matching export
                        for " + imp.getVersionConstraint());
        }
        if (imp.getSelectedSupplier() != null) {
        //对导出者进行重新画线。
                rewireBundle( ((ResolverExport)
                        imp.getSelectedSupplier()).getExporter(),
                        visited);
        }
}

//判断一个Bundle是否可以解析：是否满足单件约束、符合执行环境、平台过滤。
private boolean isResolvable(BundleDescription bundle,
        Dictionary[] platformProperties,
        ArrayList rejectedSingletons) {
        //判断是否为拒绝Singleton。
```

```
        if (rejectedSingletons.contains(bundle))
                return false;
        //检查Singletons。
        if (bundle.isSingleton()) {
                Object[] sameName =
                        resolverBundles.get(bundle.getName());
                if (sameName.length > 1) //是否已经有解析的。
                        for (int i = 0; i < sameName.length; i++) {
                                if (sameName[i] == bundle ||
                                        !((ResolverBundle) sameName[i]).
                                        getBundle().isSingleton())
                                        continue; //忽略。
                                if (((ResolverBundle) sameName[i]).
                                getBundle().isResolved()) {
                                        rejectedSingletons.add(bundle);
                                        return false; //缓存。
                                }
                        }
        }
        //检查环境属性。
        ……

        //检查平台过滤。
        ……
}

//附加片段到Host。解决Host约束，并附加片段。
private void attachFragment(ResolverBundle bundle,
        ArrayList rejectedSingletons) {
        if (!bundle.isFragment() || !bundle.isResolvable() ||
                rejectedSingletons.contains(bundle.getBundle()))
                return;

        boolean foundMatch = false;
        BundleConstraint hostConstraint = bundle.getHost();
        Object[] hosts = resolverBundles.get(
                hostConstraint.getVersionConstraint().getName());
```

```
		for (int i = 0; i < hosts.length; i++)
			if (((ResolverBundle) hosts[i]).isResolvable() &&
				hostConstraint.isSatisfiedBy(
					(ResolverBundle) hosts[i])) {
				foundMatch = true;
				resolverExports.put(((ResolverBundle)
					hosts[i]).attachFragment(bundle, true));
			}
		if (!foundMatch)
			state.addResolverError(bundle.getBundle(),
			ResolverError.MISSING_FRAGMENT_HOST,
			bundle.getHost().getVersionConstraint().toString(),
			bundle.getHost().getVersionConstraint());
	}
	//刷新指定Bundle。
	public synchronized void resolve(BundleDescription[] reRefresh,
		Dictionary[] platformProperties) {
		if (state == null)
			throw new IllegalStateException("RESOLVER_NO_STATE");

		if (!initialized)
			initialize();
		developmentMode = .....
		reRefresh = addDevConstraints(reRefresh);
		//Unresolve这些Bundle。
		if (reRefresh != null)
			for (int i = 0; i < reRefresh.length; i++) {
				ResolverBundle rb = (ResolverBundle)
					bundleMapping.get(reRefresh[i]);
				if (rb != null)
					unresolveBundle(rb, false);
					//删除Export等，并从Resolved-Pool删除。
			}
		//重新排序。
		resolverExports.reorder();
		resolverBundles.reorder();
		resolverGenerics.reorder();
```

```
getCurrentEEs(platformProperties);
ArrayList rejectedSingletons = new ArrayList();
boolean resolveOptional = platformProperties.length == 0 ?
      false : "true".equals(
      platformProperties[0].get("osgi.resolveOptional"));
ResolverBundle[] currentlyResolved = null;
if (resolveOptional) {
      BundleDescription[] resolvedBundles =
            state.getResolvedBundles();
      currentlyResolved =
            new ResolverBundle[resolvedBundles.length];
      for (int i = 0; i < resolvedBundles.length; i++)
            currentlyResolved[i] = (ResolverBundle)
                  bundleMapping.get(resolvedBundles[i]);
}
//尝试解析未解析的Bundle。
ResolverBundle[] bundles =
      (ResolverBundle[]) unresolvedBundles.toArray(
      new ResolverBundle[unresolvedBundles.size()]);
resolveBundles(bundles, platformProperties,
      rejectedSingletons);
if (selectSingletons(bundles, rejectedSingletons)) {
      //选择一个已经解析的单件。
      bundles =
            (ResolverBundle[]) unresolvedBundles.toArray(
            new ResolverBundle[unresolvedBundles.size()]);
      resolveBundles(bundles, platformProperties,
            rejectedSingletons);
}
for (Iterator rejected = rejectedSingletons.iterator();
      rejected.hasNext();) {
      BundleDescription reject = (BundleDescription)
            rejected.next();
      BundleDescription sameName =
           state.getBundle(reject.getSymbolicName(),  null);
      state.addResolverError(reject,
            ResolverError.SINGLETON_SELECTION,
```

```
                    sameName.toString(), null);
        }
        if (resolveOptional)
              resolveOptionalConstraints(currentlyResolved);
  }
  //开发模式的限制。
  private BundleDescription[] addDevConstraints(
        BundleDescription[] reRefresh) {
        ……
  }
  //将片段约束添加到Hosts。
  private void addHostsFromFragmentConstraints(
        ResolverBundle unresolved, Set additionalRefresh) {
        if (!unresolved.isFragment())
              return;
        ImportPackageSpecification[] newImports =
              unresolved.getBundle().getImportPackages();
        BundleSpecification[] newRequires =
              unresolved.getBundle().getRequiredBundles();
        if (newImports.length == 0 && newRequires.length == 0)
              return; //片段没有约束。
        BundleConstraint hostConstraint = unresolved.getHost();
        Object[] hosts = resolverBundles.get(
              hostConstraint.getVersionConstraint().getName());
        for (int j = 0; j < hosts.length; j++)
              if (hostConstraint.isSatisfiedBy(
                    (ResolverBundle) hosts[j]) &&
                    ((ResolverBundle) hosts[j]).isResolved())
                    //需要刷新Host。
                    additionalRefresh.add(
                          ((ResolverBundle) hosts[j]).getBundle());

  }
  //解析可选约束。
  private void resolveOptionalConstraints(ResolverBundle[] bundles)
  {
        for (int i = 0; i < bundles.length; i++) {
```

```
                if (bundles[i] != null)
                        resolveOptionalConstraints(bundles[i]);
        }
}

//这个方法不正确。
private void resolveOptionalConstraints(ResolverBundle bundle) {
        BundleConstraint[] requires = bundle.getRequires();
        ArrayList cycle = new ArrayList();
        boolean resolvedOptional = false;
        for (int i = 0; i < requires.length; i++)
                if (requires[i].isOptional() &&
                        requires[i].getSelectedSupplier() == null) {
                        cycle.clear();
                        resolveRequire(requires[i], cycle);
                        if (requires[i].getSelectedSupplier() != null)
                                resolvedOptional = true;
                }
        ResolverImport[] imports = bundle.getImportPackages();
        for (int i = 0; i < imports.length; i++)
                if (imports[i].isOptional() &&
                        imports[i].getSelectedSupplier() == null) {
                        cycle.clear();
                        resolveImport(imports[i], cycle);
                        if (imports[i].getSelectedSupplier() != null)
                                resolvedOptional = true;
                }
        if (resolvedOptional) { //这个有问题，对于Optional，不需要从解析
                //池中删除，只需要重新解析Optional-Import。
                state.resolveBundle(bundle.getBundle(), false, null,
                        null, null, null);
                stateResolveConstraints(bundle);
                stateResolveBundle(bundle);
        }
}

private void getCurrentEEs(Dictionary[] platformProperties) {
        .....
```

```
}
//解析Bundles。
private void resolveBundles(ResolverBundle[] bundles,
      Dictionary[] platformProperties,
      ArrayList rejectedSingletons) {
      //删除错误日志。
      for (int i = 0; i < bundles.length; i++) {
            state.removeResolverErrors(bundles[i].getBundle());
            // 开发模式额外操作。
            bundles[i].setResolvable(
                  isResolvable(bundles[i].getBundle(),
                  platformProperties, rejectedSingletons) ||
                  developmentMode);
            bundles[i].clearRefs();
      }
      resolveBundles0(bundles, platformProperties,
            rejectedSingletons);
      if (DEBUG_WIRING)
            printWirings();
      //设置解析状态。
      stateResolveBundles(bundles);
}

private void resolveBundles0(ResolverBundle[] bundles,
      Dictionary[] platformProperties,
      ArrayList rejectedSingletons) {
      if (developmentMode)
            //重新排序。
            Arrays.sort(bundles);
      //附加所有的片段。
      for (int i = 0; i < bundles.length; i++)
            attachFragment(bundles[i], rejectedSingletons);

      //循环依赖检测。以小起点开始。
      ArrayList cycle = new ArrayList(1);
      //尝试解决所有未解析的Bundle。
      for (int i = 0; i < bundles.length; i++) {
```

```
                cycle.clear();
                resolveBundle(bundles[i], cycle);
                //检查是否有环。如果在环中有任何未解析的Bundle，我们需要解析。
                checkCycle(cycle);
        }
        //解析所有未解析的片段。
        if (unresolvedBundles.size() > 0) {
                ResolverBundle[] unresolved =
                        (ResolverBundle[]) unresolvedBundles.toArray(
                        new ResolverBundle[unresolvedBundles.size()]);
                for (int i = 0; i < unresolved.length; i++)
                        resolveFragment(unresolved[i]);
        }
        //检查Uses约束。
        checkUsesConstraints(bundles, platformProperties,
                rejectedSingletons);
}

//checkUsesConstraint，忽略。

//检查依赖环。
private void checkCycle(ArrayList cycle) {
        int cycleSize = cycle.size();
        if (cycleSize == 0)
                return;
        cycleLoop:
        for (Iterator iCycle = cycle.iterator(); iCycle.hasNext();)
        {
                ResolverBundle cycleBundle =
                (ResolverBundle) iCycle.next();
                if (!cycleBundle.isResolvable()) {
                        iCycle.remove(); //从需要解析的Bundle中删除。
                        continue cycleLoop;
                }

                ResolverImport[] imports =
                        cycleBundle.getImportPackages();
```

```
            for (int j = 0; j < imports.length; j++) {
                  //检查一个有效的Supplier。
                 while (imports[j].getSelectedSupplier() != null)
                  {
                        ResolverExport importSupplier =
                              (ResolverExport) imports[j].
                              getSelectedSupplier();
                        if (importSupplier.isDropped())
                              imports[j].selectNextSupplier();
                        else
                              break;
                  }
                  //Import不能解析。
                  if (!imports[j].isDynamic() &&
                        !imports[j].isOptional() &&
                        imports[j].getSelectedSupplier() == null)
                  {
                        cycleBundle.setResolvable(false);
                        cycleBundle.clearRefs();
                        state.addResolverError(
                        imports[j].getVersionConstraint().
                        getBundle(),
                        ResolverError.MISSING_IMPORT_PACKAGE,
                        imports[j].getVersionConstraint().
                        toString(),
                        imports[j].getVersionConstraint());
                        iCycle.remove();
                        continue cycleLoop;
                  }
            }
      }
      if (cycle.size() != cycleSize) {
            //因为环中有不能解析的Bundle，因此必须重新解析。
            for (int i = 0; i < cycle.size(); i++) {
                  ResolverBundle cycleBundle =
                        (ResolverBundle) cycle.get(i);
                  cycleBundle.clearWires();
                  cycleBundle.clearRefs();
```

```
                }
                //尝试解析这个环中能够解析的Bundle。
                ArrayList innerCycle = new ArrayList(cycle.size());
                for (int i = 0; i < cycle.size(); i++)
                      resolveBundle((ResolverBundle) cycle.get(i),
                      innerCycle);
                checkCycle(innerCycle);
          } else { //环中所有Bundle都能够解析。
                for (int i = 0; i < cycle.size(); i++) {
                      setBundleResolved(
                            (ResolverBundle) cycle.get(i));
                }
          }
    }
    //找到Singleton冲突的Bundle，反解析这个Bundle。
    private boolean selectSingletons(ResolverBundle[] bundles,
          ArrayList rejectedSingletons) {
          if (developmentMode)
                return false;
          boolean result = false;
          for (int i = 0; i < bundles.length; i++) {
               BundleDescription bundleDesc = bundles[i].getBundle();
                if (!bundleDesc.isSingleton()
                      || !bundleDesc.isResolved() ||
                      rejectedSingletons.contains(bundleDesc))
                      continue;
                Object[] sameName = resolverBundles.get(
                      bundleDesc.getName());
                if (sameName.length > 1) {
                      for (int j = 0; j < sameName.length; j++) {
                            BundleDescription sameNameDesc =
((VersionSupplier) sameName[j]).getBundle();
                            ResolverBundle sameNameBundle =
(ResolverBundle) sameName[j];
                            if (sameName[j] == bundles[i]
|| !sameNameDesc.isSingleton() || !sameNameDesc.isResolved() ||
rejectedSingletons.contains(sameNameDesc))
                                  continue;
```

```
                                result = true;
                                boolean rejectedPolicy =
selectionPolicy != null ? selectionPolicy.compare(sameNameDesc,
bundleDesc) < 0 :
sameNameDesc.getVersion().compareTo(bundleDesc.getVersion()) > 0;
                                int sameNameRefs =
sameNameBundle.getRefs();
                                int curRefs = bundles[i].getRefs();
                                if ((sameNameRefs == curRefs &&
rejectedPolicy) || sameNameRefs > curRefs) {
      if (!rejectedSingletons.contains(bundles[i].getBundle()))

      rejectedSingletons.add(bundles[i].getBundle());
                                      break;
                                }
                                if
(!rejectedSingletons.contains(sameNameDesc))

      rejectedSingletons.add(sameNameDesc);
                          }
                    }
              }
              //反解析这些Bundle。
              for (Iterator rejects = rejectedSingletons.iterator();
                    rejects.hasNext();)
                    unresolveBundle((ResolverBundle)
                          bundleMapping.get(rejects.next()), false);
              return result;
      }
      //解析片段。
      private void resolveFragment(ResolverBundle fragment) {
              if (!fragment.isFragment())
                    return;
              if (fragment.getHost().getNumPossibleSuppliers() > 0)
                    if (!developmentMode || state.getResolverErrors(
                          fragment.getBundle()).length == 0)
                          setBundleResolved(fragment);
      }
```

```
//这个方法将尝试解析指定Bundle和依赖的所有Bundle。
private boolean resolveBundle(ResolverBundle bundle,
      ArrayList cycle) {
      if (bundle.isFragment())
            return false;
      if (!bundle.isResolvable()) {
            return false;
      }
      switch (bundle.getState()) {
            case ResolverBundle.RESOLVED :
                  return true;
            case ResolverBundle.UNRESOLVED :
                  //设置Resolving状态。
                  bundle.clearWires();
                  setBundleResolving(bundle);
                  break;
            case ResolverBundle.RESOLVING :
                  if (cycle.contains(bundle))
                        return true;
                  break;
            default :
                  break;
      }

      boolean failed = false;

      if (!failed) {
            GenericConstraint[] genericRequires =
                  bundle.getGenericRequires();
            for (int i = 0; i < genericRequires.length; i++) {
                  if (!resolveGenericReq(
                        genericRequires[i], cycle)) {
                        state.addResolverError(……);
                        if (genericRequires[i].isFromFragment())
                        {
                              if (!developmentMode)
                                    …detach fragment.
```

```
                        continue;
                    }
                    if (!developmentMode) {
                        failed = true;
                        break;
                    }
                }
            }
        }

        if (!failed) {
            BundleConstraint[] requires = bundle.getRequires();
            for (int i = 0; i < requires.length; i++) {
                if (!resolveRequire(requires[i], cycle)) {
                    //同上，设置错误，Detach Fragment等。
                }
            }
        }

        if (!failed) {
            ResolverImport[] imports = bundle.getImportPackages();
            for (int i = 0; i < imports.length; i++) {
                //仅解析非Dynamic。
                if (!imports[i].isDynamic() &&
                    !resolveImport(imports[i], cycle)) {
                    //同上。
                }
            }
        }

        //检查片段约束。
        checkFragmentConstraints(bundle);

        //开发模式的额外检验。
        if (developmentMode && !failed &&
        state.getResolverErrors(bundle.getBundle()).length > 0)
            failed = true;
```

```
        if (failed) {
                setBundleUnresolved(bundle, false, developmentMode);
        } else if (!cycle.contains(bundle)) {
                setBundleResolved(bundle);
        }

        if (bundle.getState() == ResolverBundle.UNRESOLVED)
                bundle.setResolvable(false);

        return bundle.getState() != ResolverBundle.UNRESOLVED;
}
//检查片段约束。
private void checkFragmentConstraints(ResolverBundle bundle) {
        //获取所有的片段，冲突检测，如果冲突，则分离片段和Export。
        ……
}

private boolean resolveGenericReq(GenericConstraint constraint,
        rayList cycle) {
        //解析一般需求，忽略。
}

//解析Require-Bundle约束。
private boolean resolveRequire(BundleConstraint req,
        ArrayList cycle) {
        //如果Supplier不为空，则将依赖Bundle添加到环中。
        if (req.getSelectedSupplier() != null) {
                if (!cycle.contains(req.getBundle())) {
                        cycle.add(req.getBundle());
                }
                return true; //已经连过线。
        }
        Object[] bundles = resolverBundles.get(
                req.getVersionConstraint().getName());
        boolean result = false;
        for (int i = 0; i < bundles.length; i++) {
```

```
                ResolverBundle bundle = (ResolverBundle) bundles[i];
                //检查Import约束是否满足。
                if (req.isSatisfiedBy(bundle)) {
                        bundle.addRef(req.getBundle());//添加引用。
                        req.addPossibleSupplier(bundle);
                        if (req.getBundle() != bundle) {
                                if (bundle.getState() !=
                                        ResolverBundle.RESOLVED &&
                                        !resolveBundle(bundle, cycle)
                                         && !developmentMode) {
                                        req.removePossibleSupplier(bundle);
                                        continue; //未解析。
                                }
                        }
                        //检查循环依赖。
                        if (req.getBundle() != bundle) {
                                if (bundle.getState() ==
                                        ResolverBundle.RESOLVING)
                                        //如果Bundle是Resolving，则循环。
                                        if (!cycle.contains(
                                                req.getBundle())) {
                                                cycle.add(req.getBundle());
                                        }
                        }
                        //找到匹配项，正在连线。
                        result = true;
                }
        }
        if (result || req.isOptional())
                return true; //Optional总是返回true。

        return false;
}

//解析Import约束。
private boolean resolveImport(ResolverImport imp, ArrayList cycle)
{
```

```
if (imp.getSelectedSupplier() != null) {
      //查环。
      if (!cycle.contains(imp.getBundle())) {
            cycle.add(imp.getBundle());
      }
      return true;            }
boolean result = false;
Object[] exports = resolverExports.get(imp.getName());
exportsloop:
for (int i = 0; i < exports.length; i++) {
      ResolverExport export = (ResolverExport) exports[i];
      if (imp.isSatisfiedBy(export)) {
            int originalState =
                  export.getExporter().getState();
            if (imp.isDynamic() && originalState !=
                  ResolverBundle.RESOLVED)
                  continue; //不解析Dynamic。
            if (imp.getBundle() == export.getExporter() &&
            !export.getExportPackageDescription().isRoot())
                  continue; //不能连接到Re-export。
            if (imp.getSelectedSupplier() != null &&
                  ((ResolverExport) imp.
                        getSelectedSupplier()).getExporter()
                        == imp.getBundle())
                  break; //连接到本身，没有关系。
            export.getExporter().addRef(imp.getBundle());
            imp.addPossibleSupplier(export);
            ResolverExport[] importerExps = null;
            if (imp.getBundle() != export.getExporter()) {
                  importerExps = imp.getBundle().
                        getExports(imp.getName());
                  for (int j = 0; j < importerExps.length;
                        j++) {
                        if (importerExps[j].
                              getExportPackageDescription()
                              .isRoot() &&
                              !export.
```

```
                        getExportPackageDescription()
                        .isRoot())
                        continue exportsloop;
                        //阻止Import到Re-exports。
                  if (importerExps[j].
                        getExportPackageDescription()
                        .isRoot())
                              resolverExports.remove(
                              importerExps[j]);
                              //引用成功，删除Export。
            }
            //开发模式允许一个约束到一个为解析的Bundle。
            if ((originalState !=
                  ResolverBundle.RESOLVED &&
                   !resolveBundle(export.getExporter(
                  ), cycle) && !developmentMode) ||
                  export.isDropped()) {
                  imp.removePossibleSupplier(export);
                  for (int j = 0; j <
                        importerExps.length; j++)
                        resolverExports.put(
                        importerExps[j].getName(),
                        importerExps[j]);
                  continue; // 未解析。
            }
      } else if (export.isDropped())
            continue;
      //记录循环依赖。
      if (imp.getBundle() != export.getExporter())
            if (export.getExporter().getState() ==
                  ResolverBundle.RESOLVING) {
                  if (!cycle.contains(
                        imp.getBundle())) {
                        cycle.add(imp.getBundle());
                  }
            }
      result = true;
```

```
                }
        }

        if (result)
                return true;
        if (resolveImportReprovide(imp, cycle))
                return true;
        if (imp.isOptional())
                return true;
        return false;
}

//检查Import是否可以用Re-Export解析，找到真正的Exporter。
private boolean resolveImportReprovide(ResolverImport imp,
        ArrayList cycle) {
        String bsn = ((ImportPackageSpecification) imp.
                getVersionConstraint()).getBundleSymbolicName();

        if (bsn == null)
                return false;

        Object[] bundles = resolverBundles.get(bsn);
        for (int i = 0; i < bundles.length; i++)
             if (resolveBundle((ResolverBundle) bundles[i], cycle))
                    if (resolveImportReprovide0(imp,
                            (ResolverBundle) bundles[i],
                            (ResolverBundle) bundles[i], cycle,
                            new ArrayList(5)))
                            return true;
        return false;
}

private boolean resolveImportReprovide0(ResolverImport imp,
        ResolverBundle reexporter, ResolverBundle rb,
        ArrayList cycle, ArrayList visited) {
        if (visited.contains(rb))
                return false; //循环检测。
        visited.add(rb);
```

```
BundleConstraint[] requires = rb.getRequires();
for (int i = 0; i < requires.length; i++) {
      if (!((BundleSpecification) requires[i].
            getVersionConstraint()).isExported())
            continue;
      if (requires[i].getSelectedSupplier() == null)
            continue;
      ResolverExport[] exports =
            ((ResolverBundle) requires[i].
            getSelectedSupplier()).getExports(
            imp.getName());
      for (int j = 0; j < exports.length; j++) {
            Map directives = exports[j].
            getExportPackageDescription().getDirectives();
            directives.remove(Constants.USES_DIRECTIVE);
            ExportPackageDescription epd = state.
            getFactory().createExportPackageDescription(exp
            orts[j].getName(), exports[j].getVersion(),
            directives,
            exports[j].getExportPackageDescription().getAtt
            ributes(), false, reexporter.getBundle());
            if (imp.getVersionConstraint().
                  isSatisfiedBy(epd)) {
                  // Create reexport and add to bundle and
                  ResolverExport re = new
                        ResolverExport(reexporter, epd);
                  reexporter.addExport(re);
                  resolverExports.put(re.getName(), re);

                  imp.addPossibleSupplier(re);
                  return true;
            }
      }
      if (resolveImportReprovide0(imp, reexporter,
            (ResolverBundle) requires[i].
            getSelectedSupplier(), cycle, visited))
            return true;
}
```

```
        return false;
    }

    //将一个Bundle移到UNRESOLVED状态。
    private void setBundleUnresolved(ResolverBundle bundle,
        boolean removed, boolean keepFragsAttached) {
        if (bundle.getState() == ResolverBundle.UNRESOLVED &&
                !developmentMode)
            return;
        if (removed || !keepFragsAttached) {
            resolverExports.remove(bundle.getExportPackages());
            resolverGenerics.remove(
                    bundle.getGenericCapabilities());
            bundle.detachAllFragments();
            bundle.initialize(false);
            if (!removed) {
                resolverExports.put(bundle.getExportPackages());
                resolverGenerics.put(
                        bundle.getGenericCapabilities());
            }
        }
        if (!removed && (!developmentMode
            || !unresolvedBundles.contains(bundle)))
            unresolvedBundles.add(bundle);
        bundle.setState(ResolverBundle.UNRESOLVED);
    }

    private void setBundleResolved(ResolverBundle bundle) {
        if (bundle.getState() == ResolverBundle.RESOLVED)
            return;
        unresolvedBundles.remove(bundle);
        bundle.setState(ResolverBundle.RESOLVED);
    }

    private void setBundleResolving(ResolverBundle bundle) {
        if (bundle.getState() == ResolverBundle.RESOLVING)
            return;
        unresolvedBundles.remove(bundle);
```

```
        bundle.setState(ResolverBundle.RESOLVING);
    }

    //在系统状态中解析Bundle。
    private void stateResolveBundles(ResolverBundle[] resolvedBundles)
    {
        for (int i = 0; i < resolvedBundles.length; i++) {
            if (!resolvedBundles[i].getBundle().isResolved())
                stateResolveBundle(resolvedBundles[i]);
        }
    }
    //在State解析约束，包括Import、Require和Generics。
    private void stateResolveConstraints(ResolverBundle rb) {
        ResolverImport[] imports = rb.getImportPackages();
        for (int i = 0; i < imports.length; i++) {
            ResolverExport export =
            (ResolverExport) imports[i].getSelectedSupplier();
            BaseDescription supplier = export == null ? null :
            export.getExportPackageDescription();
            state.resolveConstraint(
                imports[i].getVersionConstraint(), supplier);
        }
        BundleConstraint[] requires = rb.getRequires();
        for (int i = 0; i < requires.length; i++) {
            ResolverBundle bundle = (ResolverBundle) requires[i].
                getSelectedSupplier();
            BaseDescription supplier = bundle == null ? null :
                bundle.getBundle();
            state.resolveConstraint(
                requires[i].getVersionConstraint(), supplier);
        }
        GenericConstraint[] genericRequires =
            rb.getGenericRequires();
        for (int i = 0; i < genericRequires.length; i++) {
            GenericCapability[] matchingCapabilities =
                genericRequires[i].getMatchingCapabilities();
            if (matchingCapabilities == null)
                state.resolveConstraint(
```

```
                    genericRequires[i].getVersionConstraint(),
                    null);
            else
                for (int j = 0; j < matchingCapabilities.length;
                    j++)
                    state.resolveConstraint(
                    genericRequires[i].getVersionConstraint(),
                    matchingCapabilities[j].
                    getBaseDescription());
        }
    }
    //记录片段约束——片段的Import和Require约束。
    private void stateResolveFragConstraints(ResolverBundle rb) {
        ResolverBundle host = (ResolverBundle) rb.getHost().
            getSelectedSupplier();
        ImportPackageSpecification[] imports =
            rb.getBundle().getImportPackages();
        for (int i = 0; i < imports.length; i++) {
            ResolverImport hostImport = host == null ? null :
                host.getImport(imports[i].getName());
            ResolverExport export = (ResolverExport) (
                hostImport == null ? null :
                hostImport.getSelectedSupplier());
            BaseDescription supplier = export == null ? null :
                export.getExportPackageDescription();
            state.resolveConstraint(imports[i], supplier);
        }
        BundleSpecification[] requires =
            rb.getBundle().getRequiredBundles();
        for (int i = 0; i < requires.length; i++) {
            BundleConstraint hostRequire = host == null ? null :
                host.getRequire(requires[i].getName());
            ResolverBundle bundle = (ResolverBundle) (
                hostRequire == null ? null :
                hostRequire.getSelectedSupplier());
            BaseDescription supplier = bundle == null ? null :
                bundle.getBundle();
            state.resolveConstraint(requires[i], supplier);
```

```
        }
    }
    //在State中解析Bundle。
    private void stateResolveBundle(ResolverBundle rb) {
        if (!rb.isResolved() && !developmentMode)
            return;
        if (rb.isFragment())
            stateResolveFragConstraints(rb);
        else
            stateResolveConstraints(rb);
        //收集exports。
        ResolverExport[] exports = rb.getSelectedExports();
        ArrayList selectedExports = new ArrayList(exports.length);
        for (int i = 0; i < exports.length; i++) {
            selectedExports.add(
                exports[i].getExportPackageDescription());
        }
        ExportPackageDescription[] selectedExportsArray =
            (ExportPackageDescription[]) selectedExports.toArray(
            new  ExportPackageDescription[selectedExports.size()]);

        //收集连线的Exports。
        ResolverImport[] imports = rb.getImportPackages();
        ArrayList exportsWiredTo = new ArrayList(imports.length);
        for (int i = 0; i < imports.length; i++)
            if (imports[i].getSelectedSupplier() != null)
                exportsWiredTo.add(
                    imports[i].getSelectedSupplier().
                    getBaseDescription());
        ExportPackageDescription[] exportsWiredToArray =
            (ExportPackageDescription[])
                exportsWiredTo.toArray(new
                ExportPackageDescription[exportsWiredTo.size()]
            );

        //收集连线的Bundle。
        BundleConstraint[] requires = rb.getRequires();
```

```
            ArrayList bundlesWiredTo = new ArrayList(requires.length);
            for (int i = 0; i < requires.length; i++)
                  if (requires[i].getSelectedSupplier() != null)
                        bundlesWiredTo.add(
                        requires[i].getSelectedSupplier().
                        getBaseDescription());
            BundleDescription[] bundlesWiredToArray =
                  (BundleDescription[]) bundlesWiredTo.toArray(
                  new BundleDescription[bundlesWiredTo.size()]);
            //片段。
            BundleDescription[] hostBundles = null;
            if (rb.isFragment()) {
                  VersionSupplier[] matchingBundles =
                        rb.getHost().getPossibleSuppliers();
                  if (matchingBundles != null &&
                        matchingBundles.length > 0) {
                        hostBundles = new BundleDescription[
                              matchingBundles.length];
                       for (int i = 0; i < matchingBundles.length; i++)
                        {
                              hostBundles[i] =
                                    matchingBundles[i].getBundle();
                              if (rb.isNewFragmentExports() &&
                                    hostBundles[i].isResolved()) {
                                    //更新Export。
                                    ResolverExport[] hostExports =
((ResolverBundle) matchingBundles[i]).getSelectedExports();
                                    ExportPackageDescription[]
hostExportsArray = new ExportPackageDescription[hostExports.length];
                                    for (int j = 0; j <
                                          hostExports.length; j++)
                                          hostExportsArray[j] =
hostExports[j].getExportPackageDescription();
                                    state.resolveBundle(hostBundles[i],
true, null, hostExportsArray, hostBundles[i].getResolvedRequires(),
hostBundles[i].getResolvedImports());
                              }
                        }
```

```
                }
            }

            //在State中解析Bundle。
            state.resolveBundle(rb.getBundle(), rb.isResolved(),
hostBundles, selectedExportsArray, bundlesWiredToArray,
exportsWiredToArray);
        }

        //解析动态Import。
        public synchronized ExportPackageDescription
            resolveDynamicImport(
                BundleDescription importingBundle,
                String requestedPackage) {
            if (state == null)
                throw new IllegalStateException("RESOLVER_NO_STATE");

            //确保解析器已经初始化。
            if (!initialized)
                initialize();

            ResolverBundle rb = (ResolverBundle)
                bundleMapping.get(importingBundle);
            if (rb.getExport(requestedPackage) != null)
                return null; //不允许动态引用的该Bundle的导出。
            ResolverImport[] resolverImports = rb.getImportPackages();

            boolean found = false;
            for (int j = 0; j < resolverImports.length; j++) {
                if (!resolverImports[j].isDynamic())
                    continue;
                String importName = resolverImports[j].getName();

                if (importName.equals("*") ||
                    (importName.endsWith(".*") &&
                    requestedPackage.startsWith(
                    importName.substring(0, importName.length() -
```

```
                        2)))) {
                resolverImports[j].setName(requestedPackage);
            }
            //解析Import。
            if (requestedPackage.equals(
                    resolverImports[j].getName())) {
                found = true;
                //收集Grouping Checker。
                ……

            }
            //清空。
            resolverImports[j].setName(null);
        }
        //支持快速添加动态Import。
        if (!found) {
            Map directives = new HashMap(1);
            directives.put(
                    Constants.RESOLUTION_DIRECTIVE,
                    ImportPackageSpecification.RESOLUTION_DYNAMIC);
            ImportPackageSpecification packageSpec = state.
                    getFactory().createImportPackageSpecification(
                    requestedPackage, null, null, null, directives,
                    null, importingBundle);
            ResolverImport newImport = new ResolverImport(rb,
                    packageSpec);
            //解析Import，不需要管Cycle。
            if (resolveImport(newImport, new ArrayList())) {
                while (newImport.getSelectedSupplier() != null)
                {
                    if (groupingChecker.isDynamicConsistent(
                            rb, (ResolverExport) newImport.
                            getSelectedSupplier()) != null)
                            newImport.selectNextSupplier();
                    else
                            break;
                }
```

```
                    return ((ResolverExport) newImport.
                          getSelectedSupplier()).
                          getExportPackageDescription();
                }
          }
          return null;
    }
    //添加一个BundleDescription。
    public void bundleAdded(BundleDescription bundle) {
          if (!initialized)
                return;

          boolean alreadyThere = false; //是否已经添加了
          for (int i = 0; i < unresolvedBundles.size(); i++) {
                ResolverBundle rb = (ResolverBundle)
                      unresolvedBundles.get(i);
                if (rb.getBundle() == bundle) {
                      alreadyThere = true;
                }
          }
          if (!alreadyThere) {
                ResolverBundle rb = new ResolverBundle(bundle, this);
                bundleMapping.put(bundle, rb);
                unresolvedBundles.add(rb);
                resolverExports.put(rb.getExportPackages());
                resolverBundles.put(rb.getName(), rb);
                resolverGenerics.put(rb.getGenericCapabilities());
          }
    }
    //从解析器中删除一个BundleDescription，删除内容包括RemovalPending列表、
    //BundleMapping列表、ExportDes仓库、BundleDes仓库、
    //UnresolvedBundle仓库。
    public void bundleRemoved(BundleDescription bundle,
          boolean pending) {
          if (pending)
                removalPending.put(
                      new Long(bundle.getBundleId()), bundle);
```

```
        if (!initialized)
                return;
        ResolverBundle rb =
                (ResolverBundle) bundleMapping.get(bundle);
        if (rb == null)
                return;

        if (!pending) {
                bundleMapping.remove(bundle);
                groupingChecker.clear(rb);
        }
        if (!pending || !bundle.isResolved()) {
                resolverExports.remove(rb.getExportPackages());
                resolverBundles.remove(rb);
                resolverGenerics.remove(rb.getGenericCapabilities());
        }
        unresolvedBundles.remove(rb);
}
//反解析Bundle。
//1 从Export、Bundle描述仓库删除；2 修改状态；3 从State中删除。
private void unresolveBundle(ResolverBundle bundle,
        boolean removed) {
        if (bundle == null)
                return;
        Object[] removedBundles = removalPending.remove(
                new Long(bundle.getBundle().getBundleId()));
        for (int i = 0; i < removedBundles.length; i++) {
                ResolverBundle re =
                (ResolverBundle)  bundleMapping.get(removedBundles[i]);
                unresolveBundle(re, true);
                state.removeBundleComplete(
                        (BundleDescription) removedBundles[i]);
                resolverExports.remove(re.getExportPackages());
                resolverBundles.remove(re);
                resolverGenerics.remove(re.getGenericCapabilities());
                bundleMapping.remove(removedBundles[i]);
                groupingChecker.clear(re);
                //判断是否需要删除。
```

```
                if (removedBundles[i] == bundle.getBundle())
                        removed = true;
        }

        if (!bundle.getBundle().isResolved() && !developmentMode)
                return;
        setBundleUnresolved(bundle, removed, false);
        BundleDescription[] dependents =
                bundle.getBundle().getDependents();
        //从State中反解析。
        state.resolveBundle(bundle.getBundle(), false, null, null,
                null, null);
        //反解析依赖的Budle。
        for (int i = 0; i < dependents.length; i++)
                unresolveBundle((ResolverBundle)
                        bundleMapping.get(dependents[i]), false);
}
//Removed然后Added。
public void bundleUpdated(BundleDescription newDescription,
        BundleDescription existingDescription, boolean pending) {
        bundleRemoved(existingDescription, pending);
        bundleAdded(newDescription);
}
//清空所有临时数据。
public void flush() {
        resolverExports = null;
        resolverBundles = null;
        resolverGenerics = null;
        unresolvedBundles = null;
        bundleMapping = null;
        Object[] removed = removalPending.getAllValues();
        for (int i = 0; i < removed.length; i++)
                state.removeBundleComplete(
                (BundleDescription) removed[i]);
        removalPending.clear();
        initialized = false;
}
```

```
	public State getState() {
		return state;
	}

	public void setState(State newState) {
		state = newState;
		flush();
	}

	//Log……

	VersionHashMap getResolverExports() {
		return resolverExports;
	}

	public void setSelectionPolicy(Comparator selectionPolicy) {
		this.selectionPolicy = selectionPolicy;
	}

	public Comparator getSelectionPolicy() {
		return selectionPolicy;
	}
}
```

3.3 osgi.eclipse.osgi.internal.resolver

3.3.1 BundleDeltaImpl

```
package org.eclipse.osgi.internal.resolver;

import org.eclipse.osgi.service.resolver.BundleDelta;
import org.eclipse.osgi.service.resolver.BundleDescription;

public class BundleDeltaImpl implements BundleDelta {
```

```
private BundleDescription bundleDescription;
private int type;

public BundleDeltaImpl(BundleDescription bundleDescription) {
      this(bundleDescription, 0);
}

public BundleDeltaImpl(BundleDescription bundleDescription,
      int type) {
      this.bundleDescription = bundleDescription;
      this.type = type;
}
//发生变化的Bundle。
public BundleDescription getBundle() {
      return bundleDescription;
}
//变更类型。
public int getType() {
      return type;
}

protected void setBundle(BundleDescription bundleDescription) {
      this.bundleDescription = bundleDescription;
}

protected void setType(int type) {
      this.type = type;
}

public String toString() {
      ……
}
//按BundleID比较。
public int compareTo(Object obj) {
      long idcomp = getBundle().getBundleId() -
            ((BundleDelta) obj).getBundle().getBundleId();
      return (idcomp < 0L) ? -1 : ((idcomp > 0L) ? 1 : 0);
}
```

```
}
```

3.3.2 StateDeltaImpl

```
package org.eclipse.osgi.internal.resolver;

import java.util.*;

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

public class StateDeltaImpl implements StateDelta {
	private State state;
	private Map changes = new HashMap();

	public StateDeltaImpl(State state) {
		this.state = state;
	}
	//返回所有发生变化的Bundle的变更情况。
	public BundleDelta[] getChanges() {
		return (BundleDelta[]) changes.values().toArray(
			new BundleDelta[changes.size()]);
	}
	//发生指定变化的Bundle变更情况。
	public BundleDelta[] getChanges(int mask, boolean exact) {
		List result = new ArrayList();
		for (Iterator changesIter = changes.values().iterator();
			changesIter.hasNext();) {
			BundleDelta change = (BundleDelta) changesIter.next();
			if (mask == change.getType() ||
				(!exact && (change.getType() & mask) != 0))
				result.add(change);
		}
		return (BundleDelta[]) result.toArray(
			new BundleDelta[result.size()]);
	}
	//关联状态。
	public State getState() {
```

```
        return state;
    }
    //记录添加一个Bundle。
    void recordBundleAdded(BundleDescriptionImpl added) {
        BundleDeltaImpl change =
                (BundleDeltaImpl) changes.get(added);
        if (change == null) { //如果原来不存在，则直接添加。
            changes.put(added, new BundleDeltaImpl(
                    added, BundleDelta.ADDED));
            return;
        }
        if (change.getType() == BundleDelta.REMOVED) {
            changes.remove(added); //如果系统状态已删除它，添加后
                                   //系统状态保持不变。
            return;
        }
        int newType = change.getType();
        if ((newType & BundleDelta.REMOVED) != 0)
            newType &= ~BundleDelta.REMOVED;
        change.setType(newType | BundleDelta.ADDED); //状态进行位或。
        change.setBundle(added);
    }
    //记录更新。
    void recordBundleUpdated(BundleDescriptionImpl updated) {
        BundleDeltaImpl change =
        (BundleDeltaImpl) changes.get(updated);
        if (change == null) {
            changes.put(updated, new BundleDeltaImpl(
            updated, BundleDelta.UPDATED));
            return;
        }
        if ((change.getType() & (BundleDelta.ADDED |
            BundleDelta.REMOVED)) != 0)
            return;
        change.setType(change.getType() | BundleDelta.UPDATED);
        change.setBundle(updated);
    }
```

```
//记录删除。
void recordBundleRemoved(BundleDescriptionImpl removed) {
	BundleDeltaImpl change =
		(BundleDeltaImpl) changes.get(removed);
	if (change == null) {
		changes.put(removed, new BundleDeltaImpl(
			removed, BundleDelta.REMOVED));
		return;
	}
	if (change.getType() == BundleDelta.ADDED) {
		changes.remove(removed);
		return;
	}
	int newType = change.getType();
	if ((newType & BundleDelta.ADDED) != 0)
		newType &= ~BundleDelta.ADDED;
	change.setType(newType | BundleDelta.REMOVED);
}
//记录正在删除。
void recordBundleRemovalPending(BundleDescriptionImpl removed) {
	BundleDeltaImpl change =
		(BundleDeltaImpl) changes.get(removed);
	if (change == null) {
		changes.put(removed, new BundleDeltaImpl(
		removed, BundleDelta.REMOVAL_PENDING));
		return;
	}
	int newType = change.getType();
	if ((newType & BundleDelta.REMOVAL_COMPLETE) != 0)
		newType &= ~BundleDelta.REMOVAL_COMPLETE;
	change.setType(newType | BundleDelta.REMOVAL_PENDING);
}
//记录删除完毕。
void recordBundleRemovalComplete(BundleDescriptionImpl removed) {
	BundleDeltaImpl change =
		(BundleDeltaImpl) changes.get(removed);
	if (change == null) {
		changes.put(removed, new BundleDeltaImpl(
```

```
                        removed, BundleDelta.REMOVAL_COMPLETE));
                return;
            }
            int newType = change.getType();
            if ((newType & BundleDelta.REMOVAL_PENDING) != 0)
                newType &= ~BundleDelta.REMOVAL_PENDING;
            change.setType(newType | BundleDelta.REMOVAL_COMPLETE);
        }
        //记录解析或为解析。
        void recordBundleResolved(BundleDescriptionImpl resolved,
            boolean result) {
            if (resolved.isResolved() == result) //如果未变更，则不记录。
                return;
            BundleDeltaImpl change =
                (BundleDeltaImpl) changes.get(resolved);
            int newType = result ? BundleDelta.RESOLVED :
                BundleDelta.UNRESOLVED;
            if (change == null) {
                change = new BundleDeltaImpl(resolved, newType);
                changes.put(resolved, change);
                return;
            }

            newType = newType | (change.getType() &
                ~(BundleDelta.RESOLVED | BundleDelta.UNRESOLVED));
            change.setType(newType);
            change.setBundle(resolved);
        }
}
```

3.3.3 StateImpl

系统状态类。

```
package org.eclipse.osgi.internal.resolver;

import java.util.*;
```

```
import org.eclipse.osgi.framework.debug.Debug;
import org.eclipse.osgi.framework.debug.FrameworkDebugOptions;
import org.eclipse.osgi.framework.internal.core.Constants;
import org.eclipse.osgi.framework.internal.core.FilterImpl;
import org.eclipse.osgi.framework.util.*;
import org.eclipse.osgi.internal.baseadaptor.StateManager;
import org.eclipse.osgi.service.resolver.*;
import org.eclipse.osgi.util.ManifestElement;
import org.osgi.framework.*;

public abstract class StateImpl implements State {
	public static final String[] PROPS = {"osgi.os", "osgi.ws",
		"osgi.nl", "osgi.arch",
		Constants.OSGI_FRAMEWORK_SYSTEM_PACKAGES,
		Constants.OSGI_RESOLVER_MODE,
		Constants.FRAMEWORK_EXECUTIONENVIRONMENT,
		"osgi.resolveOptional", "osgi.genericAliases"};
	//关联解析器。
	transient private Resolver resolver;
	//系统状态变更。
	transient private StateDeltaImpl changes;
	transient private boolean resolving = false;
	//正在删除的Bundle。
	transient private HashSet removalPendings = new HashSet();
	private boolean resolved = true;
	private long timeStamp = System.currentTimeMillis();
	//所有Bundle描述。
	private KeyedHashSet bundleDescriptions = new KeyedHashSet(false);
	//所有解析错误。
	private HashMap resolverErrors = new HashMap();
	//状态对象工厂。
	private StateObjectFactory factory;
	//已解析的Bundle。
	private KeyedHashSet resolvedBundles = new KeyedHashSet();
	boolean fullyLoaded = false;
	private boolean dynamicCacheChanged = false;
	//状态Reader，仅用于Bundle描述的晚加载。
```

```
private StateReader reader;
private Dictionary[] platformProperties = {
      new Hashtable(PROPS.length)};
//最高BundleID。
private long highestBundleId = -1;
private HashSet platformPropertyKeys = new HashSet(PROPS.length);

private static long cumulativeTime;

//阻止外部访问。
protected StateImpl() {
      addPlatformPropertyKeys(PROPS);
}
//添加Bundle到系统状态。
public boolean addBundle(BundleDescription description) {
      if (!basicAddBundle(description)) //添加到Bundle描述库。
            return false;
      String platformFilter = description.getPlatformFilter();
      if (platformFilter != null) {
            try {
                  FilterImpl filter = (FilterImpl)
                  FrameworkUtil.createFilter(platformFilter);
                  addPlatformPropertyKeys(filter.getAttributes());
            } catch (InvalidSyntaxException e) {
            }
      }
      //系统状态发生变化。
      resolved = false;
      //记录Bundle变化。
      getDelta().recordBundleAdded(
            (BundleDescriptionImpl) description);
      //是否扩展Bundle。
      if (Constants.getInternalSymbolicName().equals(
            description.getSymbolicName()))
            resetSystemExports();
      //通知解析器，添加了一个Bundle。
      if (resolver != null)
```

```
            resolver.bundleAdded(description);
        //时间戳加1。
        updateTimeStamp();
        return true;
    }

    public boolean updateBundle(BundleDescription newDescription) {
        BundleDescriptionImpl existing =
                (BundleDescriptionImpl) bundleDescriptions.get(
                        (BundleDescriptionImpl) newDescription);
        if (existing == null)
                return false;
        if (!bundleDescriptions.remove(existing))
                return false;
        resolvedBundles.remove(existing);
        existing.setStateBit(BundleDescriptionImpl.REMOVAL_PENDING,
                true);
        if (!basicAddBundle(newDescription))
                return false;
        resolved = false;
        getDelta().recordBundleUpdated(
                (BundleDescriptionImpl) newDescription);
        if (Constants.getInternalSymbolicName().equals(
                newDescription.getSymbolicName()))
                resetSystemExports();
        if (resolver != null) {
             boolean pending = existing.getDependents().length > 0;
                resolver.bundleUpdated(
                        newDescription, existing, pending);
                if (pending) {
                        getDelta().recordBundleRemovalPending(existing);
                        removalPendings.add(existing);
                } else {
                        synchronized (this) {
                                try {
                                        resolving = true;
                                        resolverErrors.remove(existing);
                                       resolveBundle(existing, false, null,
```

```
                                        null, null, null);
                            } finally {
                                  resolving = false;
                            }
                      }
                }
          }
          updateTimeStamp();
          return true;
    }
    //删除一个指定ID的Bundle。
    public BundleDescription removeBundle(long bundleId) {
          BundleDescription toRemove = getBundle(bundleId);
          if (toRemove == null || !removeBundle(toRemove))
                return null;
          return toRemove;
    }
    //删除一个Bundle。
    public boolean removeBundle(BundleDescription toRemove) {
          if (!bundleDescriptions.remove((KeyedElement) toRemove))
                return false;
          resolvedBundles.remove((KeyedElement) toRemove);
          resolved = false;
          //记录删除一个Bundle变更。
          getDelta().recordBundleRemoved(
                (BundleDescriptionImpl) toRemove);
          ((BundleDescriptionImpl) toRemove).setStateBit(
                BundleDescriptionImpl.REMOVAL_PENDING, true);
          if (resolver != null) {
               boolean pending = toRemove.getDependents().length > 0;
                //通知解析器，一个Bundle已经被删除。
                //如果有依赖的话，转入正在删除。
                resolver.bundleRemoved(toRemove, pending);
                if (pending) {
                      getDelta().recordBundleRemovalPending(
                            (BundleDescriptionImpl) toRemove);
                      removalPendings.add(toRemove);
```

```
                } else {
                        synchronized (this) {
                                try {
                                        resolving = true;
                                        resolverErrors.remove(toRemove);
                                        //反解析Bundle。
                                       resolveBundle(toRemove, false, null,
                                                null, null, null);
                                } finally {
                                        resolving = false;
                                }
                        }
                }
        }
        updateTimeStamp();
        return true;
}

public StateDelta getChanges() {
        return getDelta();
}

private StateDeltaImpl getDelta() {
        if (changes == null)
                changes = new StateDeltaImpl(this);
        return changes;
}
//Bundle搜索。
public BundleDescription[] getBundles(String symbolicName) {
        if (Constants.OSGI_SYSTEM_BUNDLE.equals(symbolicName))
                symbolicName = Constants.getInternalSymbolicName();
        final List bundles = new ArrayList();
        for (Iterator iter = bundleDescriptions.iterator();
                iter.hasNext();) {
                BundleDescription bundle =
                        (BundleDescription) iter.next();
                if (symbolicName.equals(bundle.getSymbolicName()))
                        bundles.add(bundle);
```

```
        }
        return (BundleDescription[]) bundles.toArray(
                new BundleDescription[bundles.size()]);
}

public BundleDescription[] getBundles() {
        return (BundleDescription[]) bundleDescriptions.
                elements(new BundleDescription[
                bundleDescriptions.size()]);
}

public BundleDescription getBundle(long id) {
        BundleDescription result = (BundleDescription)
                 bundleDescriptions.getByKey(new Long(id));
        if (result != null)
                return result;
        for (Iterator iter = removalPendings.iterator();
                iter.hasNext();) {
                BundleDescription removedBundle =
                        (BundleDescription) iter.next();
                if (removedBundle.getBundleId() == id)
                        return removedBundle;
        }
        return null;
}

public BundleDescription getBundle(String name, Version version)
{
        BundleDescription[] allBundles = getBundles(name);
        if (allBundles.length == 1)
                return version == null || allBundles[0].getVersion().
                        equals(version) ? allBundles[0] : null;

        if (allBundles.length == 0)
                return null;

        BundleDescription unresolvedFound = null;
        BundleDescription resolvedFound = null;
```

```
		for (int i = 0; i < allBundles.length; i++) {
			BundleDescription current = allBundles[i];
			BundleDescription base;

			if (current.isResolved())
				base = resolvedFound;
			else
				base = unresolvedFound;

			if (version == null || current.getVersion().
				equals(version)) {
				if (base != null && (base.getVersion().
					compareTo(current.getVersion()) <= 0 ||
					base.getBundleId() >
					current.getBundleId())) {
					if (base == resolvedFound)
						resolvedFound = current;
					else
						unresolvedFound = current;
				} else {
					if (current.isResolved())
						resolvedFound = current;
					else
						unresolvedFound = current;
				}

			}
		}
		if (resolvedFound != null)
			return resolvedFound;

		return unresolvedFound;
	}
	//获取时间戳。
	public long getTimeStamp() {
		return timeStamp;
	}
	//是否系统状态发生变化。
```

```
public boolean isResolved() {
	return resolved || isEmpty();
}
//利用指定Supplier解析指定约束。
public void resolveConstraint(VersionConstraint constraint,
	BaseDescription supplier) {
	((VersionConstraintImpl) constraint).setSupplier(supplier);
}
//利用指定的Supplier解析指定约束。
public synchronized void resolveBundle(BundleDescription bundle,
	boolean status, BundleDescription[] hosts,
	ExportPackageDescription[] selectedExports,
	BundleDescription[] resolvedRequires,
	ExportPackageDescription[] resolvedImports) {
	if (!resolving) //如果当前状态不是Resolving，则异常。
		throw new IllegalStateException();

	BundleDescriptionImpl modifiable =
		(BundleDescriptionImpl) bundle;
	//记录Resolved。
	getDelta().recordBundleResolved(modifiable, status);
	//晚加载为NOT。
	modifiable.setLazyLoaded(false);
	//状态为RESOLVED。
	modifiable.setStateBit(BundleDescriptionImpl.RESOLVED,
		status);
	if (status) {
		resolverErrors.remove(modifiable);
		resolvedBundles.add(modifiable);
	} else {
		//从解析池删除该Bundle。
		resolvedBundles.remove(modifiable);
		modifiable.removeDependencies();
	}
	//解析或反解析所有的Export、Require和Import。
	if (selectedExports == null || resolvedRequires == null ||
		resolvedImports == null)
```

```
                unresolveConstraints(modifiable);
        else
                resolveConstraints(modifiable, hosts, selectedExports,
                resolvedRequires, resolvedImports);
}
//删除一个Bundle完成。
public synchronized void removeBundleComplete(
        BundleDescription bundle) {
        if (!resolving)
                throw new IllegalStateException();
        //记录变更。
        getDelta().recordBundleRemovalComplete(
                (BundleDescriptionImpl) bundle);
        removalPendings.remove(bundle);
}
//为Bundle设置所有的约束和依赖。
private void resolveConstraints(BundleDescriptionImpl bundle,
        BundleDescription[] hosts,
        ExportPackageDescription[] selectedExports,
        BundleDescription[] resolvedRequires,
        ExportPackageDescription[] resolvedImports) {

        HostSpecificationImpl hostSpec =
                (HostSpecificationImpl) bundle.getHost();
        if (hostSpec != null) {
                if (hosts != null) {
                        hostSpec.setHosts(hosts);
                        for (int i = 0; i < hosts.length; i++)
                                ((BundleDescriptionImpl) hosts[i]).
                                        addDependency(bundle, true);
                }
        }

        bundle.setSelectedExports(selectedExports);
        bundle.setResolvedRequires(resolvedRequires);
        bundle.setResolvedImports(resolvedImports);

        bundle.addDependencies(hosts, true);
```

```
            bundle.addDependencies(resolvedRequires, true);
            bundle.addDependencies(resolvedImports, true);

            GenericSpecification[] genericRequires =
                  bundle.getGenericRequires();
            if (genericRequires.length > 0) {
                  ArrayList genericSuppliers =
                        new ArrayList(genericRequires.length);
                  for (int i = 0; i < genericRequires.length; i++) {
                        GenericDescription[] suppliers =
                              genericRequires[i].getSuppliers();
                        if (suppliers != null)
                             for (int j = 0; j < suppliers.length; j++)
                                   genericSuppliers.add(suppliers[j]);
                  }
                  bundle.addDependencies((BaseDescription[])
                        genericSuppliers.toArray(new BaseDescription[
                        genericSuppliers.size()]), true);
            }
      }
      //清空Bundle的所有约束和依赖。
      private void unresolveConstraints(BundleDescriptionImpl bundle) {
            HostSpecificationImpl host =
                  (HostSpecificationImpl) bundle.getHost();
            if (host != null)
                  host.setHosts(null);

            bundle.setSelectedExports(null);
            bundle.setResolvedImports(null);
            bundle.setResolvedRequires(null);

            GenericSpecification[] genericRequires =
                  bundle.getGenericRequires();
            if (genericRequires.length > 0)
                  for (int i = 0; i < genericRequires.length; i++)
                        ((GenericSpecificationImpl)
                              genericRequires[i]).setSupplers(null);
            bundle.removeDependencies();
```

```
}
//解析，incremental声明是否为渐进的方式来解析。如果是渐进防止，则只是解析
//指定的Bundle，否则将清空所有的Bundle，然后解析指定Bundle。
private synchronized StateDelta resolve(boolean incremental,
      BundleDescription[] reResolve) {
      try {
            resolving = true; //正在解析标志。
            if (resolver == null)
                  throw new IllegalStateException(
                  "no resolver set");
            fullyLoad();
            long start = 0;
            if (StateManager.DEBUG_PLATFORM_ADMIN_RESOLVER)
                  start = System.currentTimeMillis();
            //非渐进模式，则清空所有已解析的缓存。
            if (!incremental) {
                  resolved = false;
                  reResolve = getBundles();
                  //合并正在删除的Bundle。
                  if (removalPendings.size() > 0) {
                        BundleDescription[] removed =
                              getRemovalPendings();
                        reResolve = mergeBundles(
                              reResolve, removed);
                  }
                  //清空这些Bundle所有信息。
                  flush(reResolve);
            }
            if (resolved && reResolve == null)
                  return new StateDeltaImpl(this);
            if (removalPendings.size() > 0) {
                  BundleDescription[] removed =
                        getRemovalPendings();
                  reResolve = mergeBundles(reResolve, removed);
            }
            //平台属性。
            Headers[] tmpPlatformProperties = new Headers[
```

```
                    platformProperties.length];
            for (int i = 0; i < platformProperties.length; i++) {
                    tmpPlatformProperties[i] = new Headers(
                            platformProperties[i].size());
                    for (Enumeration keys = platformProperties[i].
                            keys(); keys.hasMoreElements();) {
                            Object key = keys.nextElement();
                            tmpPlatformProperties[i].put(key,
                                    platformProperties[i].get(key));
                    }
            }
            //解析这些Bundle。
            resolver.resolve(reResolve, tmpPlatformProperties);
            resolved = removalPendings.size() == 0;

            StateDelta savedChanges = changes == null ?
                    new StateDeltaImpl(this) : changes;
            changes = new StateDeltaImpl(this);
            //更新时间戳。
            if (savedChanges.getChanges().length > 0)
                    updateTimeStamp();
            return savedChanges;
        } finally {
                resolving = false; //解析完成。
        }
}
//合并Bundle。
private BundleDescription[] mergeBundles(
        BundleDescription[] reResolve, BundleDescription[] removed)
{
        if (reResolve == null) //为空，则返回正在删除的Bundel。
                return removed;
        if (reResolve.length == 0) //阻止正在删除的Bundle。
                return reResolve;
        //合并结果。
        ArrayList result = new ArrayList(reResolve.length +
                removed.length);
```

```
		for (int i = 0; i < reResolve.length; i++)
			result.add(reResolve[i]);
		for (int i = 0; i < removed.length; i++) {
			boolean found = false;
			for (int j = 0; j < reResolve.length; j++) {
				if (removed[i] == reResolve[j]) {
					found = true;
					break;
				}
			}
			if (!found)
				result.add(removed[i]);
		}
		return (BundleDescription[]) result.toArray(
			new BundleDescription[result.size()]);
	}
	//清空指定Bundle。
	private void flush(BundleDescription[] bundles) {
		resolver.flush();//刷新Resolver。
		resolved = false;
		resolverErrors.clear();
		if (resolvedBundles.isEmpty())
			return;
		//反解析已存在的Bundle。
		for (int i = 0; i < bundles.length; i++) {
			resolveBundle(bundles[i], false, null, null,
				null, null);
		}
		resolvedBundles.clear();
	}

	public StateDelta resolve() {
		return resolve(true, null);
	}

	public StateDelta resolve(boolean incremental) {
		return resolve(incremental, null);
	}
```

```
public StateDelta resolve(BundleDescription[] reResolve) {
	return resolve(true, reResolve);
}

public void setOverrides(Object value) {
	throw new UnsupportedOperationException();
}
//返回所有已经解析的Bundle。
public BundleDescription[] getResolvedBundles() {
	return (BundleDescription[]) resolvedBundles.elements(
		new BundleDescription[resolvedBundles.size()]);
}

public boolean isEmpty() {
	return bundleDescriptions.isEmpty();
}

void setResolved(boolean resolved) {
	this.resolved = resolved;
}
//仅添加一个Bundle。
boolean basicAddBundle(BundleDescription description) {
	((BundleDescriptionImpl) description).
		setContainingState(this); //设置所属的State。
	((BundleDescriptionImpl) description).setStateBit(
		BundleDescriptionImpl.REMOVAL_PENDING, false);
	if (bundleDescriptions.add(
		(BundleDescriptionImpl) description)) {
		if (description.getBundleId() > getHighestBundleId())
			highestBundleId = description.getBundleId();
		return true;
	}
	return false;
}

void addResolvedBundle(BundleDescriptionImpl resolvedBundle) {
	resolvedBundles.add(resolvedBundle);
```

```
}
//返回所有已经解析的Selected-Export，包括正在删除的Bundle。
public ExportPackageDescription[] getExportedPackages() {
	fullyLoad();
	final List allExportedPackages = new ArrayList();
	for (Iterator iter = resolvedBundles.iterator();
		iter.hasNext();) {
		BundleDescription bundle =
			(BundleDescription) iter.next();
		ExportPackageDescription[] bundlePackages =
			bundle.getSelectedExports();
		if (bundlePackages == null)
			continue;
		for (int i = 0; i < bundlePackages.length; i++)
			allExportedPackages.add(bundlePackages[i]);
	}
	for (Iterator iter = removalPendings.iterator();
		iter.hasNext();) {
		BundleDescription bundle =
			(BundleDescription) iter.next();
		ExportPackageDescription[] bundlePackages =
			bundle.getSelectedExports();
		if (bundlePackages == null)
			continue;
		for (int i = 0; i < bundlePackages.length; i++)
			allExportedPackages.add(bundlePackages[i]);
	}
	return (ExportPackageDescription[]) allExportedPackages.
		toArray(new ExportPackageDescription[
		allExportedPackages.size()]);
}
//获取一个Host的所有片段。
BundleDescription[] getFragments(final BundleDescription host) {
	final List fragments = new ArrayList();
	for (Iterator iter = bundleDescriptions.iterator();
		iter.hasNext();) {
		BundleDescription bundle =
			(BundleDescription) iter.next();
```

```
                HostSpecification hostSpec = bundle.getHost();

                if (hostSpec != null) {
                     BundleDescription[] hosts = hostSpec.getHosts();
                      if (hosts != null)
                            for (int i = 0; i < hosts.length; i++)
                                  if (hosts[i] == host) {
                                        fragments.add(bundle);
                                        break;
                                  }
                }
          }
          return (BundleDescription[]) fragments.toArray(
                new BundleDescription[fragments.size()]);
    }
    //设置和获取时间戳。
    public void setTimeStamp(long newTimeStamp) {
          timeStamp = newTimeStamp;
    }

    private void updateTimeStamp() {
          if (getTimeStamp() == Long.MAX_VALUE)
                setTimeStamp(0);
          setTimeStamp(getTimeStamp() + 1);
    }
    //设置和获取状态对象工厂。
    public StateObjectFactory getFactory() {
          return factory;
    }

    void setFactory(StateObjectFactory factory) {
          this.factory = factory;
    }
    //Bundle查询。
    public BundleDescription getBundleByLocation(String location) {
          for (Iterator i = bundleDescriptions.iterator();
                i.hasNext();) {
                BundleDescription current =
```

```
                    (BundleDescription) i.next();
                if (location.equals(current.getLocation()))
                    return current;
        }
        return null;
}

public Resolver getResolver() {
        return resolver;
}

public void setResolver(Resolver newResolver) {
        if (resolver == newResolver)
                return;
        if (resolver != null) {
                Resolver oldResolver = resolver;
                resolver = null;
                oldResolver.setState(null);
        }
        resolver = newResolver;
        if (resolver == null)
                return;
        resolver.setState(this);
}

public boolean setPlatformProperties(
        Dictionary platformProperties) {
        return setPlatformProperties(
                new Dictionary[] {platformProperties});
}

public boolean setPlatformProperties(
                Dictionary[] platformProperties) {
        return setPlatformProperties(platformProperties, true);
}

synchronized boolean setPlatformProperties(
```

```
        Dictionary[] platformProperties, boolean resetSystemExports)
{
        ……
}
//重置系统Bundle的Export。
private void resetSystemExports() {
        BundleDescription[] systemBundles = getBundles(
                Constants.getInternalSymbolicName());
        if (systemBundles.length > 0) {
                BundleDescriptionImpl systemBundle =
                        (BundleDescriptionImpl) systemBundles[0];
                ExportPackageDescription[] exports =
                        systemBundle.getExportPackages();
                ArrayList newExports = new ArrayList(exports.length);
                for (int i = 0; i < exports.length; i++)
                        if (((Integer) exports[i].getDirective(
                                ExportPackageDescriptionImpl.EQUINOX_EE))
                                        .intValue() < 0)
                                newExports.add(exports[i]);
                addSystemExports(newExports);
                systemBundle.setExportPackages(
                        (ExportPackageDescription[])
                newExports.toArray(new ExportPackageDescription[
                        newExports.size()]));
        }
}
//添加系统Export。
private void addSystemExports(ArrayList exports) {
        for (int i = 0; i < platformProperties.length; i++)
                try {
                        ManifestElement[] elements = ManifestElement.
                                parseHeader(Constants.EXPORT_PACKAGE,
                                (String) platformProperties[i].get(
                                Constants.
                                OSGI_FRAMEWORK_SYSTEM_PACKAGES));
                        if (elements == null)
                                continue;
                        ExportPackageDescription[] systemExports =
```

```
                        StateBuilder.createExportPackages(
                        elements, null, null, null, 2, false);
                Integer profInx = new Integer(i);
                for (int j = 0; j < systemExports.length; j++)
                {
                        ((ExportPackageDescriptionImpl)
                                systemExports[j]).setDirective(
                                ExportPackageDescriptionImpl.
                                EQUINOX_EE, profInx);
                        exports.add(systemExports[j]);
                }
            } catch (BundleException e) {

            }
}

public Dictionary[] getPlatformProperties() {
      return platformProperties;
}

private boolean checkProp(Object origObj, Object newObj) {
      if ((origObj == null && newObj != null) ||
            (origObj != null && newObj == null))
            return true;
      if (origObj == null)
            return false;
      if (origObj.getClass() != newObj.getClass())
            return true;
      if (origObj instanceof String)
            return !origObj.equals(newObj);
      String[] origProps = (String[]) origObj;
      String[] newProps = (String[]) newObj;
      if (origProps.length != newProps.length)
            return true;
      for (int i = 0; i < origProps.length; i++) {
            if (!origProps[i].equals(newProps[i]))
                  return true;
      }
```

```
        return false;
    }

    private boolean changedProps(Dictionary origProps,
        Dictionary newProps, String[] keys) {
        for (int i = 0; i < keys.length; i++) {
            Object origProp = origProps.get(keys[i]);
            Object newProp = newProps.get(keys[i]);
            if (checkProp(origProp, newProp))
                return true;
        }
        return false;
    }
    //返回正在删除的Bundle。
    public BundleDescription[] getRemovalPendings() {
        return (BundleDescription[]) removalPendings.toArray(
            new BundleDescription[removalPendings.size()]);
    }
    //解析动态Import。利用时间戳，这是一个好方法。
    public synchronized ExportPackageDescription linkDynamicImport(
        BundleDescription importingBundle,
        String requestedPackage) {
        if (resolver == null)
            throw new IllegalStateException("no resolver set");
        BundleDescriptionImpl importer =
            (BundleDescriptionImpl) importingBundle;
        if (importer.getDynamicStamp(requestedPackage) ==
            getTimeStamp())
            return null;
        try {
            resolving = true;
            fullyLoad();
            ExportPackageDescriptionImpl result =
                (ExportPackageDescriptionImpl)
                resolver.resolveDynamicImport(
                importingBundle, requestedPackage);
```

```
                if (result == null) //如果没有解析成功，则设置解析失败的
                                    //时间戳，这样不过时间戳不变则，总是解析失败。
                        importer.setDynamicStamp(requestedPackage,
                                new Long(getTimeStamp()));
                else {
                        importer.setDynamicStamp(requestedPackage,
                                null); //清空时间戳。
                        importer.addDynamicResolvedImport(result);
                }
                setDynamicCacheChanged(true);
                return result;
        } finally {
                resolving = false;
        }
}

void setReader(StateReader reader) {
        this.reader = reader;
}

StateReader getReader() {
        return reader;
}
//利用Reader加载。
public void fullyLoad() {
        if (reader == null)
                return;
        synchronized (reader) {
                if (fullyLoaded == true)
                        return;
                if (reader.isLazyLoaded())
                        reader.fullyLoad();
                fullyLoaded = true;
        }
}

public void unloadLazyData(long expireTime) {
```

```
		synchronized (reader) {
			if (reader.getAccessedFlag()) {
				reader.setAccessedFlag(false);
				return;
			}
			fullyLoaded = false;
			BundleDescription[] bundles = getBundles();
			for (int i = 0; i < bundles.length; i++)
				((BundleDescriptionImpl) bundles[i]).unload();
		}
	}

	public ExportPackageDescription[] getSystemPackages() {
		ArrayList result = new ArrayList();
		BundleDescription[] systemBundles =
			getBundles(Constants.getInternalSymbolicName());
		if (systemBundles.length > 0) {
			BundleDescriptionImpl systemBundle =
				(BundleDescriptionImpl) systemBundles[0];
			ExportPackageDescription[] exports =
				systemBundle.getExportPackages();
			for (int i = 0; i < exports.length; i++)
				if (((Integer) exports[i].getDirective(
					ExportPackageDescriptionImpl.EQUINOX_EE))
					.intValue() >= 0)
					result.add(exports[i]);
		}
		return (ExportPackageDescription[]) result.toArray(
			new ExportPackageDescription[result.size()]);
	}

	boolean inStrictMode() {
		return Constants.STRICT_MODE.equals(
			getPlatformProperties()[0].get(
			Constants.OSGI_RESOLVER_MODE));
	}
	//获取解析错误。
	public synchronized ResolverError[] getResolverErrors(
```

```
        BundleDescription bundle) {
        if (bundle.isResolved())
                return new ResolverError[0];
        ArrayList result = (ArrayList) resolverErrors.get(bundle);
        return result == null ? new ResolverError[0] :
                (ResolverError[]) result.toArray(
                        new ResolverError[result.size()]);
}

public synchronized void addResolverError(
        BundleDescription bundle, int type, String data,
        VersionConstraint unsatisfied) {
        if (!resolving)
                throw new IllegalStateException();
        ArrayList errors = (ArrayList) resolverErrors.get(bundle);
        if (errors == null) {
                errors = new ArrayList(1);
                resolverErrors.put(bundle, errors);
        }
        errors.add(new ResolverErrorImpl(
                (BundleDescriptionImpl) bundle, type, data,
                unsatisfied));
}

public synchronized void removeResolverErrors(B
        undleDescription bundle) {
        if (!resolving)
                throw new IllegalStateException();
        resolverErrors.remove(bundle);
}

public boolean dynamicCacheChanged() {
        return dynamicCacheChanged;
}

void setDynamicCacheChanged(boolean dynamicCacheChanged) {
        this.dynamicCacheChanged = dynamicCacheChanged;
}
```

```
	public StateHelper getStateHelper() {
		return StateHelperImpl.getInstance();
	}

	void addPlatformPropertyKeys(String[] keys) {
		synchronized (platformPropertyKeys) {
			for (int i = 0; i < keys.length; i++)
				if (!platformPropertyKeys.contains(keys[i]))
					platformPropertyKeys.add(keys[i]);
		}
	}

	String[] getPlatformPropertyKeys() {
		synchronized (platformPropertyKeys) {
			return (String[]) platformPropertyKeys.toArray(
			new String[platformPropertyKeys.size()]);
		}
	}

	public long getHighestBundleId() {
		return highestBundleId;
	}
}
```

3.3.4 ReadOnlyState

对 State 中会引起 Bundle 变更的所有方法重写，抛出异常。

```
package org.eclipse.osgi.internal.resolver;

import java.util.Dictionary;

import org.eclipse.osgi.service.resolver.*;
import org.osgi.framework.BundleException;
import org.osgi.framework.Version;

public class ReadOnlyState implements State {
```

```
private State target;

public ReadOnlyState(State target) {
	this.target = target;
}

public boolean addBundle(BundleDescription description) {
	throw new UnsupportedOperationException();
}

public StateDelta compare(State state) throws BundleException {
	return target.compare(state);
}

public BundleDescription getBundle(long id) {
	return target.getBundle(id);
}

public BundleDescription getBundle(String symbolicName,
	Version version) {
	return target.getBundle(symbolicName, version);
}

public BundleDescription getBundleByLocation(String location) {
	return target.getBundleByLocation(location);
}

public BundleDescription[] getBundles() {
	return target.getBundles();
}

public BundleDescription[] getBundles(String symbolicName) {
	return target.getBundles(symbolicName);
}

public StateDelta getChanges() {
	return target.getChanges();
}
```

```
public ExportPackageDescription[] getExportedPackages() {
	return target.getExportedPackages();
}

public StateObjectFactory getFactory() {
	return target.getFactory();
}

public BundleDescription[] getResolvedBundles() {
	return target.getResolvedBundles();
}

public long getTimeStamp() {
	return target.getTimeStamp();
}

public boolean isEmpty() {
	return target.isEmpty();
}

public boolean isResolved() {
	return target.isResolved();
}

public boolean removeBundle(BundleDescription bundle) {
	throw new UnsupportedOperationException();
}

public BundleDescription removeBundle(long bundleId) {
	throw new UnsupportedOperationException();
}

public StateDelta resolve() {
	throw new UnsupportedOperationException();
}

public StateDelta resolve(boolean incremental) {
```

```
        throw new UnsupportedOperationException();
    }

    public StateDelta resolve(BundleDescription[] discard) {
        throw new UnsupportedOperationException();
    }

    public void setOverrides(Object value) {
        throw new UnsupportedOperationException();
    }

    public boolean updateBundle(BundleDescription newDescription) {
        throw new UnsupportedOperationException();
    }

    public void resolveConstraint(VersionConstraint constraint,
        BaseDescription supplier) {
        throw new UnsupportedOperationException();
    }

    public void resolveBundle(BundleDescription bundle,
        boolean status, BundleDescription[] host,
        ExportPackageDescription[] selectedExports,
        BundleDescription[] resolvedRequires,
        ExportPackageDescription[] resolveImports) {
        throw new UnsupportedOperationException();
    }

    public void removeBundleComplete(BundleDescription bundle) {
        throw new UnsupportedOperationException();
    }

    public Resolver getResolver() {
        return null;
    }

    public void setResolver(Resolver value) {
        throw new UnsupportedOperationException();
```

```
}

public boolean setPlatformProperties(
      Dictionary platformProperties) {
      throw new UnsupportedOperationException();
}

public boolean setPlatformProperties(
      Dictionary platformProperties[]) {
      throw new UnsupportedOperationException();
}

public Dictionary[] getPlatformProperties() {
      return target.getPlatformProperties();
}

public ExportPackageDescription
      linkDynamicImport(BundleDescription importingBundle,
      String requestedPackage) {
      throw new UnsupportedOperationException();
}

public void setTimeStamp(long timeStamp) {
      throw new UnsupportedOperationException();
}

public ExportPackageDescription[] getSystemPackages() {
      return target.getSystemPackages();
}

public void addResolverError(BundleDescription bundle, int type,
      String data, VersionConstraint unsatisfied) {
      throw new UnsupportedOperationException();
}

public ResolverError[] getResolverErrors(BundleDescription bundle)
{
      return target.getResolverErrors(bundle);
```

```
	}

	public void removeResolverErrors(BundleDescription bundle) {
		throw new UnsupportedOperationException();
	}

	public StateHelper getStateHelper() {
		return StateHelperImpl.getInstance();
	}

	public long getHighestBundleId() {
		return target.getHighestBundleId();
	}
}
```

3.3.5 SyetemState

提供对解析、添加、删除、更新的同步访问。

```
package org.eclipse.osgi.internal.resolver;

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

public class SystemState extends StateImpl {
	synchronized public boolean addBundle(
		BundleDescription description) {
		return super.addBundle(description);
	}

	synchronized public boolean removeBundle(
		BundleDescription toRemove) {
		return super.removeBundle(toRemove);
	}

	synchronized public boolean updateBundle(
		BundleDescription newDescription) {
		return super.updateBundle(newDescription);
```

```
	}

	public StateDelta compare(State state) throws BundleException
	{
		throw new UnsupportedOperationException();
	}
}
```

3.3.6 UserState

保留所有 Bundle 的变更记录。

```
package org.eclipse.osgi.internal.resolver;

import java.util.HashSet;
import java.util.Set;
import org.eclipse.osgi.service.resolver.*;
import org.osgi.framework.BundleException;

public class UserState extends StateImpl {
	//这不是一个精确的记录Bundle变更的方法。
	private Set updated = new HashSet();

	public synchronized boolean removeBundle(
		BundleDescription description) {
		if (description.getLocation() != null)
			updated.remove(description.getLocation());
		if (!super.removeBundle(description))
			return false;
		return true;
	}

	public boolean updateBundle(BundleDescription newDescription) {
		if (!super.updateBundle(newDescription))
			return false;
		updated.add(newDescription.getLocation());
		return true;
	}
```

```
	public StateDelta compare(State baseState)
		throws BundleException {
		BundleDescription[] current = this.getBundles();
		StateDeltaImpl delta = new StateDeltaImpl(this);
		//处理添加和更新。
		for (int i = 0; i < current.length; i++) {
			BundleDescription existing =
				baseState.getBundleByLocation(
				current[i].getLocation());
			if (existing == null)
				delta.recordBundleAdded(
					(BundleDescriptionImpl) current[i]);
			else if (updated.contains(current[i].getLocation()))
				delta.recordBundleUpdated(
					(BundleDescriptionImpl) current[i]);
		}
		//处理删除。
		BundleDescription[] existing = baseState.getBundles();
		for (int i = 0; i < existing.length; i++) {
			BundleDescription local = getBundleByLocation(
				existing[i].getLocation());
			if (local == null)
				delta.recordBundleRemoved(
					(BundleDescriptionImpl) existing[i]);
		}
		return delta;
	}
}
```

4 Related Classes

4.1 StateManager

实现 PlatformAdmin 服务，管理框架的系统状态。

```
package org.eclipse.osgi.internal.baseadaptor;
```

```
import java.io.File;
import java.io.IOException;
import org.eclipse.osgi.framework.internal.core.FrameworkProperties;
import org.eclipse.osgi.internal.resolver.*;
import org.eclipse.osgi.service.resolver.*;
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleException;

public class StateManager implements PlatformAdmin, Runnable {
	//状态晚加载标识。
	public static String PROP_NO_LAZY_LOADING =
		"osgi.noLazyStateLoading";
	//设置清理晚加载数据前的间隔。
	public static String PROP_LAZY_UNLOADING_TIME =
		"osgi.lazyStateUnloadingTime";
	private long expireTime = 300000; //默认为5分钟。
	private long readStartupTime;
	private StateImpl systemState;
	private StateObjectFactoryImpl factory;
	private long lastTimeStamp;
	private boolean cachedState = false;
	private File stateFile;
	private File lazyFile;
	private long expectedTimeStamp;
	private BundleContext context;
	private Thread dataManagerThread;

	//构建一个实例。StateFile是持久化的数据，LazyFile是用于晚加载的数据，可以从
	//内存中清洗掉。
	public StateManager(File stateFile, File lazyFile,
		BundleContext context) {
		//-1表示不检测时间戳。
		this(stateFile, lazyFile, context, -1);
	}

	public StateManager(File stateFile, File lazyFile,
		BundleContext context, long expectedTimeStamp) {
```

```
        this.stateFile = stateFile;
        this.lazyFile = lazyFile;
        this.context = context;
        this.expectedTimeStamp = expectedTimeStamp;
        factory = new StateObjectFactoryImpl();
    }
    //关闭StateManager。
    public void shutdown(File stateFile, File lazyFile)
        throws IOException {
        //解析删除的Bundle。
        BundleDescription[] removalPendings =
                systemState.getRemovalPendings();
        if (removalPendings.length > 0)
                systemState.resolve(removalPendings);
        //持久化数据。
        writeState(systemState, stateFile, lazyFile);
        //停止。
        stopDataManager();
    }
    //更新持久数据。TODO:需要考虑更新晚加载数据。
    public void update(File stateFile, File lazyFile)
        throws IOException {
        BundleDescription[] removalPendings =
                systemState.getRemovalPendings();
        StateImpl state = systemState;
        if (removalPendings.length > 0) {
                state = (StateImpl) state.getFactory().
                        createState(systemState);
                state.setResolver(getResolver(
                        System.getSecurityManager() != null));
                state.setPlatformProperties(
                        FrameworkProperties.getProperties());
                state.resolve(false);
        }
        //写文件。
        writeState(state, stateFile, lazyFile);
        lastTimeStamp = state.getTimeStamp();
```

```
}

private void readSystemState(File stateFile, File lazyFile,
      long expectedTimeStamp) {
      if (stateFile == null || !stateFile.isFile())
            return;
      try {
            boolean lazyLoad = !Boolean.valueOf(
                  FrameworkProperties.getProperty(
                  PROP_NO_LAZY_LOADING)).booleanValue();
            systemState = factory.readSystemState(
                  stateFile, lazyFile, lazyLoad,
                  expectedTimeStamp);
            if (systemState == null || !initializeSystemState())
            {
                  systemState = null;
                  return;
            }
            cachedState = true;
            try {
                 expireTime = Long.parseLong(FrameworkProperties.
                        getProperty(PROP_LAZY_UNLOADING_TIME,
                        Long.toString(expireTime)));
            } catch (NumberFormatException nfe) {
                  expireTime = 0;
            }
            if (lazyLoad && expireTime > 0)
                  startDataManager();
      } catch (IOException ioe) {
            ioe.printStackTrace();
      } finally {
            //记录调试时间。
      }
}

private synchronized void startDataManager() {
      if (dataManagerThread != null)
            return;
```

```
		dataManagerThread = new Thread(this, "State Data Manager");
		dataManagerThread.setDaemon(true);
		dataManagerThread.start();
	}

	public synchronized void stopDataManager() {
		if (dataManagerThread == null)
			return;
		dataManagerThread.interrupt();
		dataManagerThread = null;
	}
	//将State写到持久文件。
	private void writeState(StateImpl state, File stateFile,
		File lazyFile) throws IOException {
		if (state == null)
			return;
		if (cachedState && !saveNeeded())
			return;
		state.fullyLoad();
		factory.writeState(state, stateFile, lazyFile);
	}

	private boolean initializeSystemState() {
		systemState.setResolver(
			getResolver(System.getSecurityManager() != null));
		lastTimeStamp = systemState.getTimeStamp();
		return !systemState.setPlatformProperties(
			FrameworkProperties.getProperties());
	}

	public synchronized State createSystemState() {
		if (systemState == null) {
			systemState = factory.createSystemState();
			initializeSystemState();
		}
		return systemState;
	}
```

```
	public synchronized State readSystemState() {
		if (systemState == null)
			readSystemState(stateFile, lazyFile,
					expectedTimeStamp);
		return systemState;
	}

	public State getSystemState() {
		return systemState;
	}

	public long getCachedTimeStamp() {
		return lastTimeStamp;
	}

	public boolean saveNeeded() {
		return systemState.getTimeStamp() != lastTimeStamp ||
			systemState.dynamicCacheChanged();
	}

	public State getState(boolean mutable) {
		return mutable ? factory.createState(systemState) :
			new ReadOnlyState(systemState);
	}

	public State getState() {
		return getState(true);
	}

	public StateObjectFactory getFactory() {
		return factory;
	}
	//不支持。
	public synchronized void commit(State state) throws
BundleException {
		throw new IllegalArgumentException("PlatformAdmin.commit()
			not supported");
	}
```

```
	public Resolver getResolver() {
		return getResolver(false);
	}

	private Resolver getResolver(boolean checkPermissions) {
		return new org.eclipse.osgi.internal.module.ResolverImpl(
				context, checkPermissions);
	}

	public StateHelper getStateHelper() {
		return StateHelperImpl.getInstance();
	}

	public void run() {
		long timeStamp = lastTimeStamp;
		while (true) {
			try {
				Thread.sleep(expireTime);
			} catch (InterruptedException e) {
				return;
			}
			if (systemState != null)
				synchronized (systemState) {
					if (timeStamp ==
systemState.getTimeStamp() && !systemState.dynamicCacheChanged())
						systemState.unloadLazyData(
						expireTime);
				}
		}
	}
}
```