

OSGI

By ZhuLinfeng

背景概览

模块层

生命周期层

服务层

背景概览

- OSGI : Open Service Gateway initiative
名字只是前身，不是重点
- OSGI 是**Java模块化的标准**，其对于动态化的支持能够帮助开发者更好地实现**即插即用**，**热部署**及**即删即无**的系统。



- OSGI 只是一个标准，就和Unix一样
- 实现：Equinox, Felix, Spring Dynamic Modules
- 应用：Eclipse , ...

实现目标

java 模块化

更高级的模块化：

即插即用

热部署

即删即无

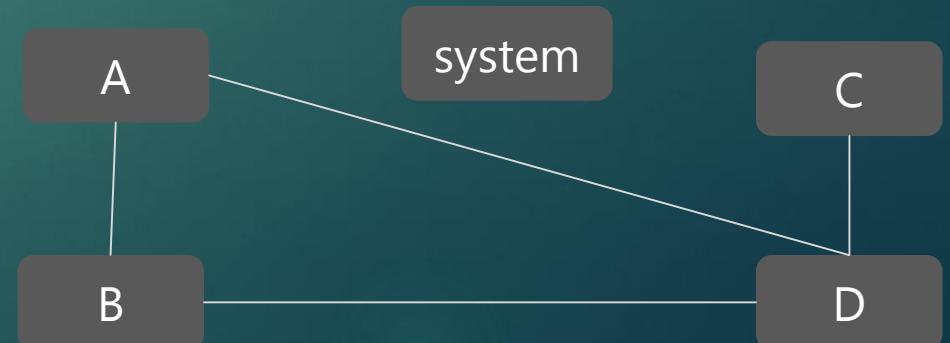
分别由OSGI三层体系架构得以满足
模块层，生命周期层，服务层

背景概览：模块概念

模块：利用一组逻辑上独立的组件集合设计出完整的系统，这些逻辑上独立的组件可称为模块。模块定义了强制性的逻辑边界，代码要么是模块的一部分，要么不是模块的一部分。模块内部的细节只对模块内部代码可见，而其他代码只能看到模块明确公开的部分（公共API）。

- API 可见 : module.h
- 实现不可见 : module.dll

- 物理上一个模块的代码捆绑在一起
- 逻辑上一个模块只负责单一的功能



背景概览：模块化的优点

高内聚

低耦合

- 功能单一
- 资源集中

- 耦合是必要的，但要少
- 通过恰如其分的暴露 API

□ 模块复用

□ 更新升级容易

□ 易维护（问题不扩散至系统）

背景概览：C 语言模块化（天然支持）

a.h

```
define SIZE 100  
void public_func();
```

module.h

```
define ...  
int func1();  
char func2();
```

接口

a.c

```
static int g_iGlobal;
```

```
static void func1( ){ ... }  
static void func2( ){ ... }
```

```
void public_func( ){ ... }
```

a.h

b.h

c.h

a.cpp

b.cpp

c.cpp

module.dll

实现

背景概览：C 语言模块化（天然支持）

- 既支持细粒度的模块化（文件级）
- 也支持粗粒度的模块化（功能模块）

没听说过有专门的 C 语言模块化框架

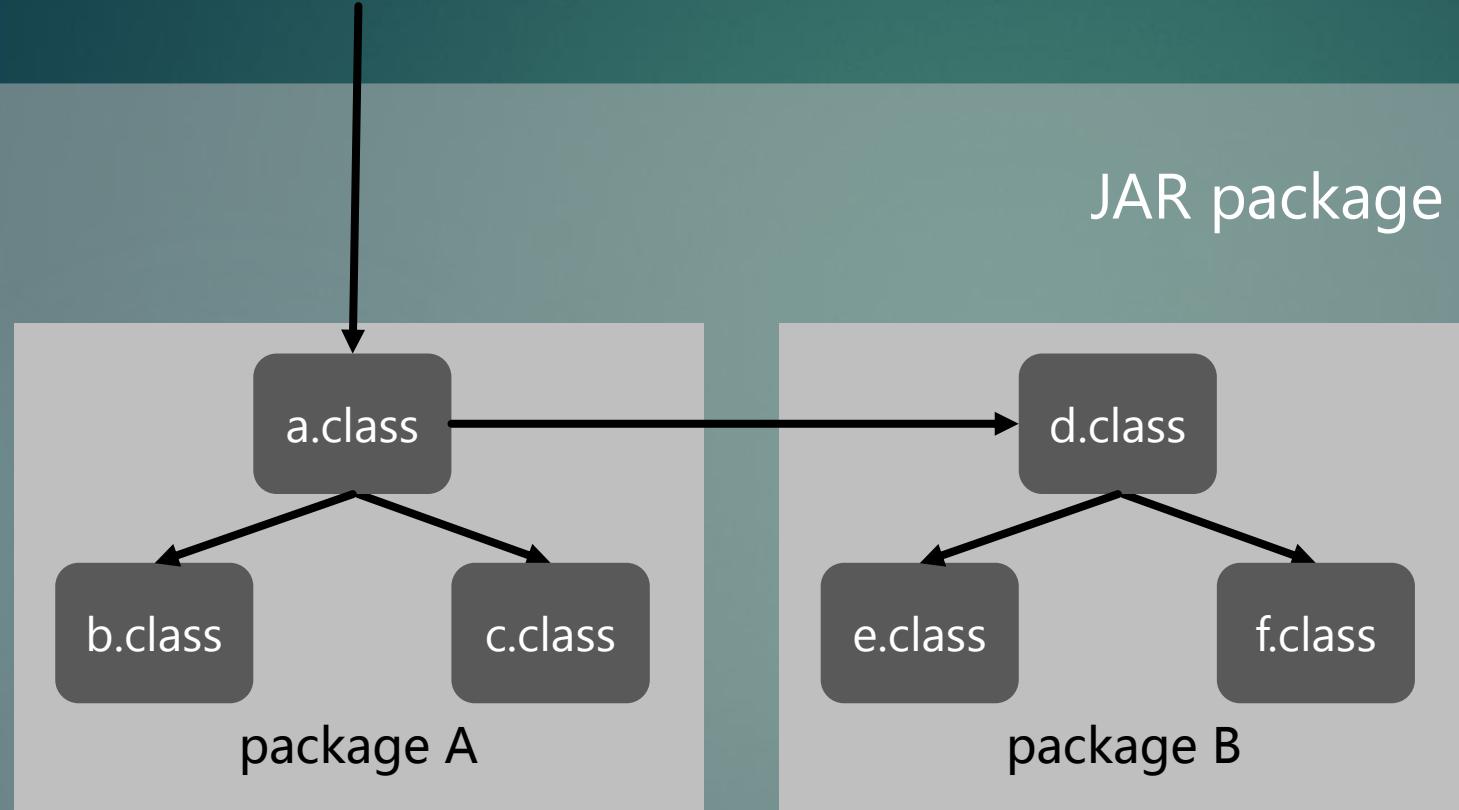
背景概览：java 语言

```
public class Bird{  
    private wings;  
    private feet;  
  
    private flap(){ ... }  
    public fly(){ ... }  
    public quack(){ ... }  
}
```

可见接口

对象本身支持很好的对象级模块化

背景概览：java 语言



开发者本想把JAR包作为一个模块，

其中会包含若干个 package， 把

a.class 作为供外界访问的唯一入

口。但根本无法阻止三方去实例

化其中任一public对象。

背景概览：java 语言

- 支持细粒度模块化（对象级）
- 对粗粒度模块化支持不够。JAR包只是简单把相关代码物理上集中在一起，但此JAR包的逻辑界限并不清晰。

需要在java基础上额外提供模块化的特性：OSGI框架

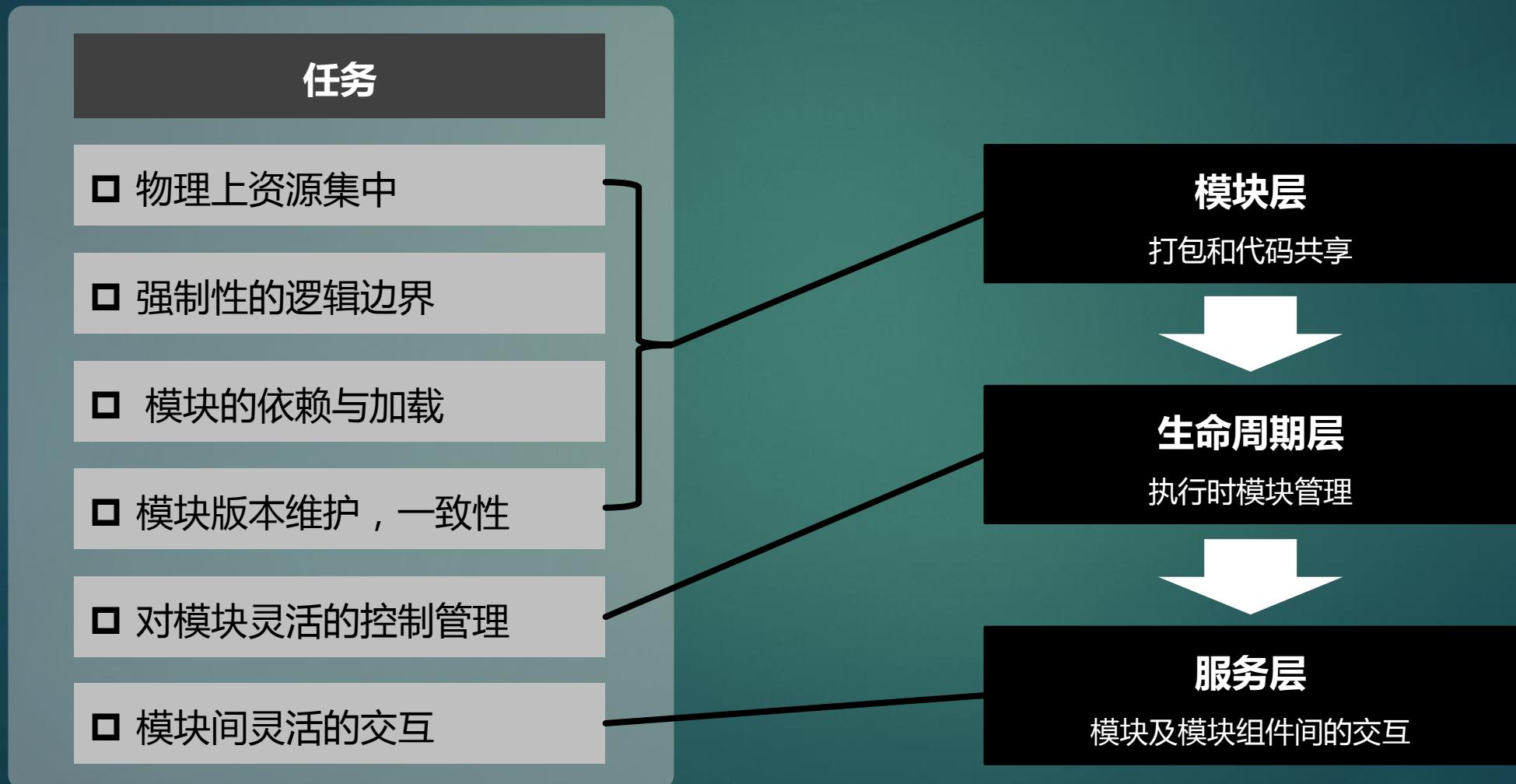
java 模块化三大不足

□ 低层代码可见性控制

□ 易错的类路径概念

□ 部署和管理支持上的不足

背景概览：OSGI 模块化任务



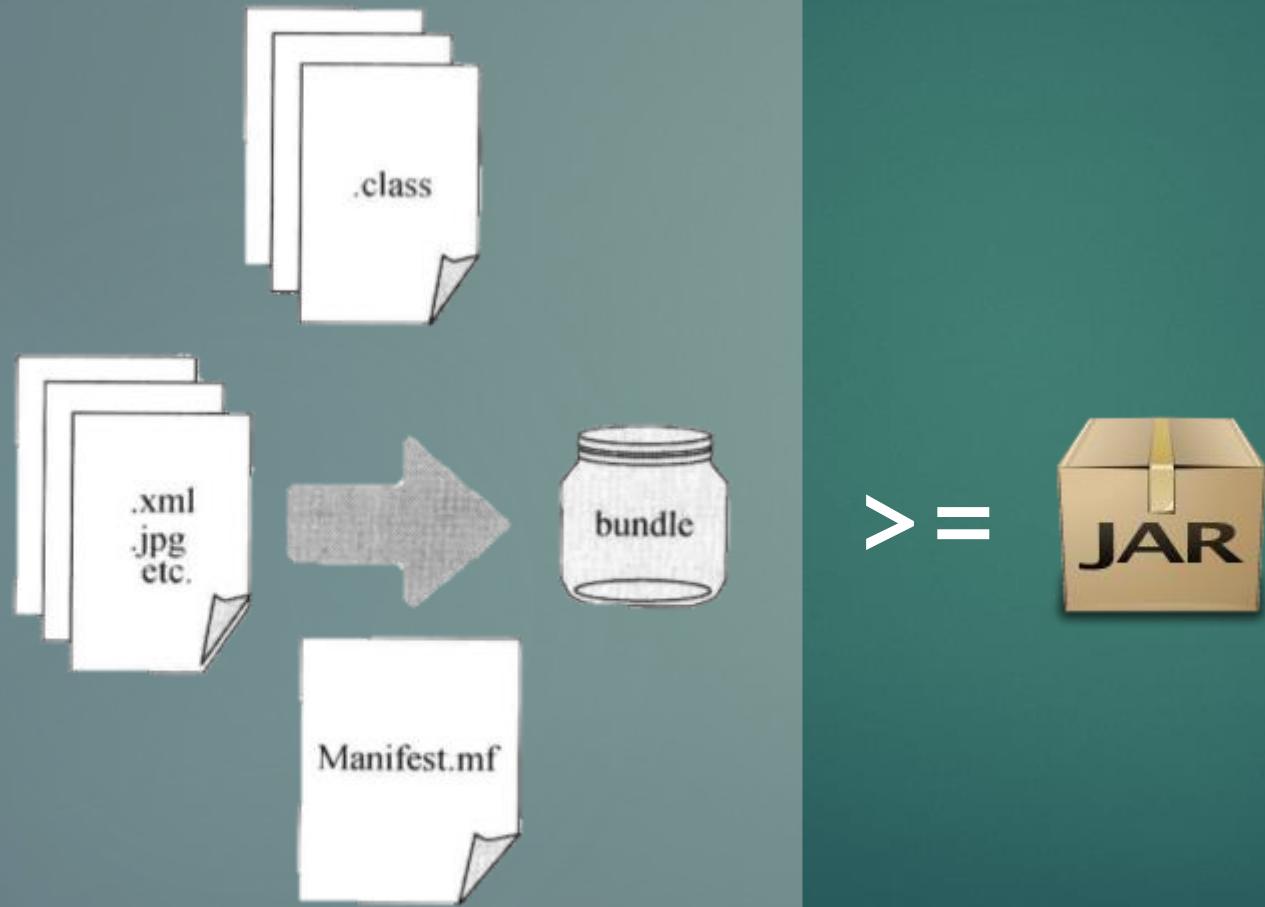


OSGI 如何通过三层体系满足java模块化特性？

模块层



模块层：bundle—物理上资源集中



bundle组成

- 类文件
- 资源文件
- 元数据

模块层：bundle—强制性的逻辑边界

MANIFEST.MF

Manifest-Version: 1.0

Built-By: some guy

Bundle-SymbolicName: xxx-toaster-provider

Bundle-Version: 1.1.0.Helium

Export-Package: toaster_provider, ...

...

普通 jar 包
其全部public类对外可见

以包为单位导出
一个bundle可以有若干个包

可以同时导出多个包

bundle 只有导出包对外可见

模块层：bundle—模块的依赖

MANIFEST.MF

Manifest-Version: 1.0

Built-By: some guy

Bundle-SymbolicName: xxx-toaster-provider

Bundle-Version: 1.1.0.Helium

Export-Package: toaster-provider

Import-Package:toaster-sample

...

OSGI自动使 java.* 包全局可见
不需由bundle显示引入

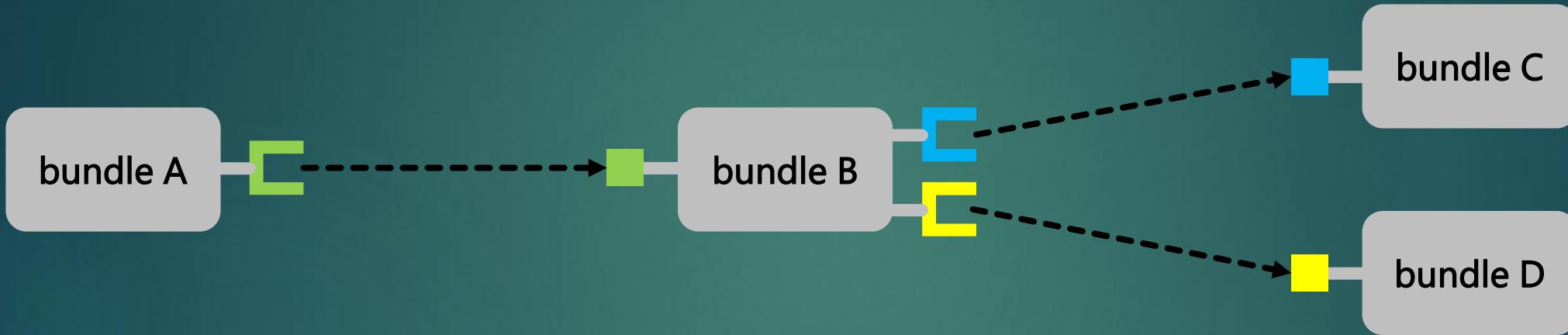
A bundle 的 import-package 要与
B bundle 的 export-package 一致

依赖是针对于导出包的，而非bundle。两个完全
不同的bundle也可以导出相同的包而被依赖。

导出包

引入其他bundle的导出包

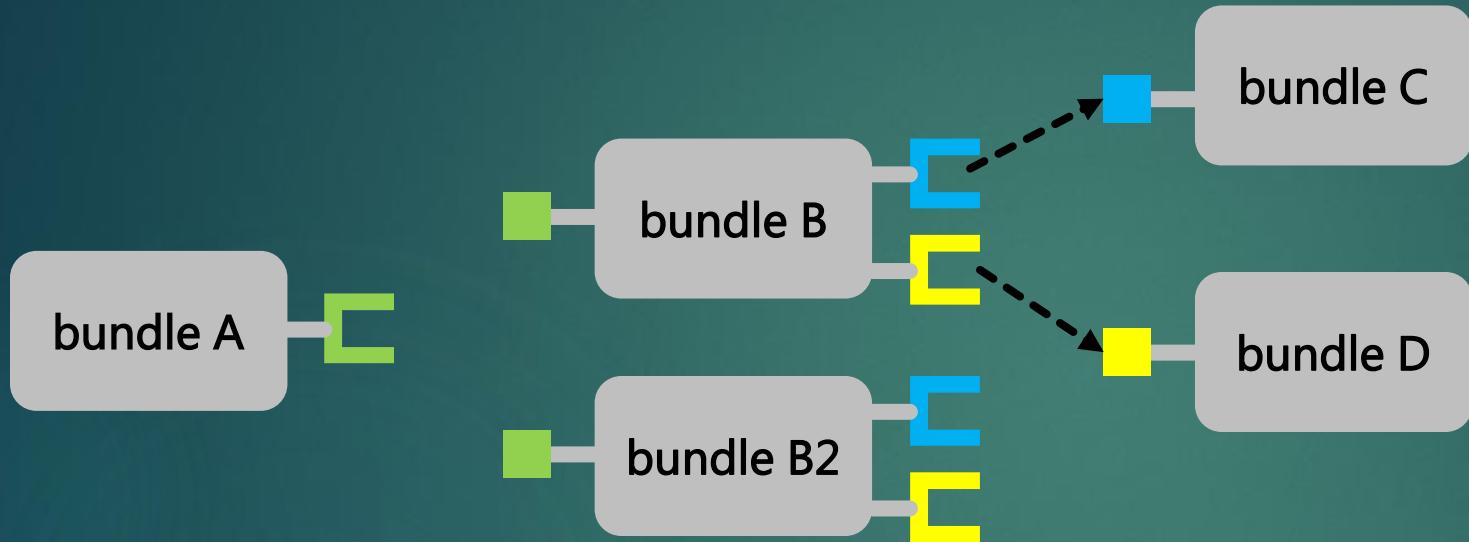
模块层：bundle—模块的依赖



- 前提是这些 bundle 已经安装在 OSGI 之中
- 依赖会传递。当要使用bundle A时，会对 A 的依赖进行解析，只有整个依赖链解析成功，A 以及依赖链上其他 bundle 才可被放心使用。
- 依赖解析结果会保存在 OSGI 中，有很多用处

解析就是将 A 的import 和 B 的export 进行匹配的过程。

模块层：bundle—模块依赖的一致性



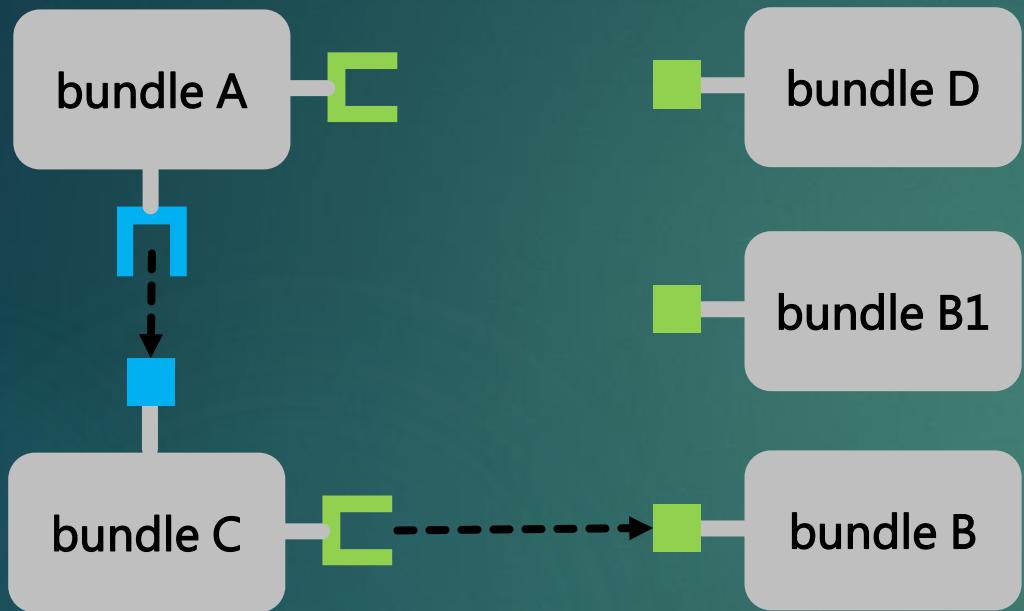
□ 当被依赖的包其bundle存在多个版本时，bundle A 如何抉择？

选择原则

1. 已解析过的bundle有最高优先级。
当有多个匹配项时，按照先版本号后安装顺序的方式确定优先级。
2. 其次是未解析过的bundle。当有多个候选项时，按照先版本号后安装顺序的方式确定优先级。

所以你觉得这里bundle A应该选谁呢？

模块层：bundle—模块依赖的一致性



- C 先被安装，并解析好了依赖，它依赖于 B；这时 A 后被安装，它对于蓝色包的依赖解析成功，是 C。但它对绿色包的依赖解析时，发现 B1 可能是最好的选择。这导致 A 和 C 对同一个包的依赖不一致，无法协同工作。

使用约束

1. bundle C 在其清单文件使用 uses:= "green"
告诉所有依赖于 C 且依赖于 green 包的 bundle 必须顺从 C 对于 green 包的依赖的选择。
2. 当解析 bundle A 时，哪怕其对 green 包的依赖有更好地选择，譬如 B1，也只得放弃而服从 C 的选择。

模块层：bundle—加载

- 当需要使用 JAR 包的一个类时，java标准做法是搜索这整个 JAR 包。
- bundle可以在清单文件中指定 Bundle-classpath，告诉 OSGI 搜索哪些路径，先后顺序如何。如不指定，也全局搜索此bundle。此为bundle的类路径。
Bundle-classpath: . , dir1/, dir2/
- 执行时，每个bundle都有个与之关联的类加载器，这个类加载器使bundle可获得其有权访问的所有类（由解析过程确定的类）。当导入bundle连接到导出bundle时，导入bundle的类加载器会得到导出bundle类加载器的引用，因此可委托其搜索导出包中的类。

类加载步骤

1. bundle执行时需要一个类
2. 如果包含该类的包名以 java. 开头，则当前类加载器的父类加载器会搜索这个类。找不到就异常结束。
3. 如果这个类在导入包中，则从导出bundle中搜索这个类。找不到就异常结束。
4. 从bundle类路径搜索，找不到就异常结束。

模块层：一些小问题



看了许久，难道对模块层就没有疑惑吗？



□ bundle 可不可以依赖自身？



□ 确实，bundle 可以导入自己的导出包。



□ 若bundle间的依赖形成环状，OSGI如何处理？



□ 我还不知道也。



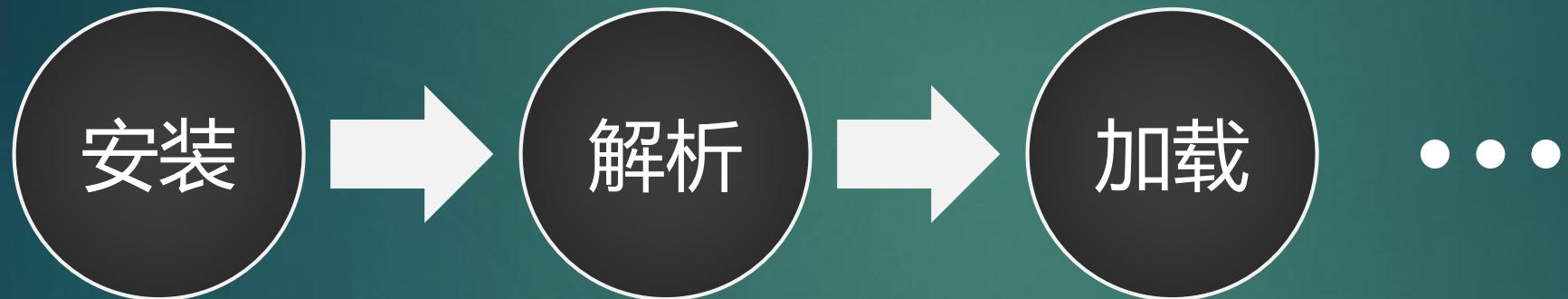
□ 看了前面的介绍，我知道bundle依赖解析是件很复杂的活，而且对于一个bundle，必须其整个依赖链全部满足才得以执行。那么若此依赖链中某个bundle发生变化，比如不见了，会怎么样？假如此依赖链很庞大，会不会很惨？



□ 这个问题超出了模块层范围，答案在生命周期层和服务层。

生命周期层中，处理方式并未超出你的预期，就是bundle大动荡。先所有bundle退回到最初状态，然后重新解析尝试恢复原来状态。在服务层中，则压根不用考虑这个问题。

模块层：bundle—生命周期雏形



欲知更多，请看后续 OSGI 之 **生命周期层** >>>

生命周期层



生命周期层：生命周期



- 软件的生命周期

为什么有生命周期

- 铁打的营盘流水的兵，一个平台要能运行无数未知的应用，应用就必须有生命周期。
- 地球要养活无数的人，人就得生老病死。
- 软件的生命周期是针对操作系统而言的。
- bundle的生命周期是针对OSGI而言的。

生命周期层：两大承诺

OSGI 生命周期层提供了两大承诺

准确定义的生命周期过程
生命周期状态图

一套操控bundle生命周期
的API

前者是实现原理，后者是公开接口，二者相对应

□ 与操作系统管理整个应用程

序不同，OSGI生命周期层允许

以bundle为单位对应用程序的

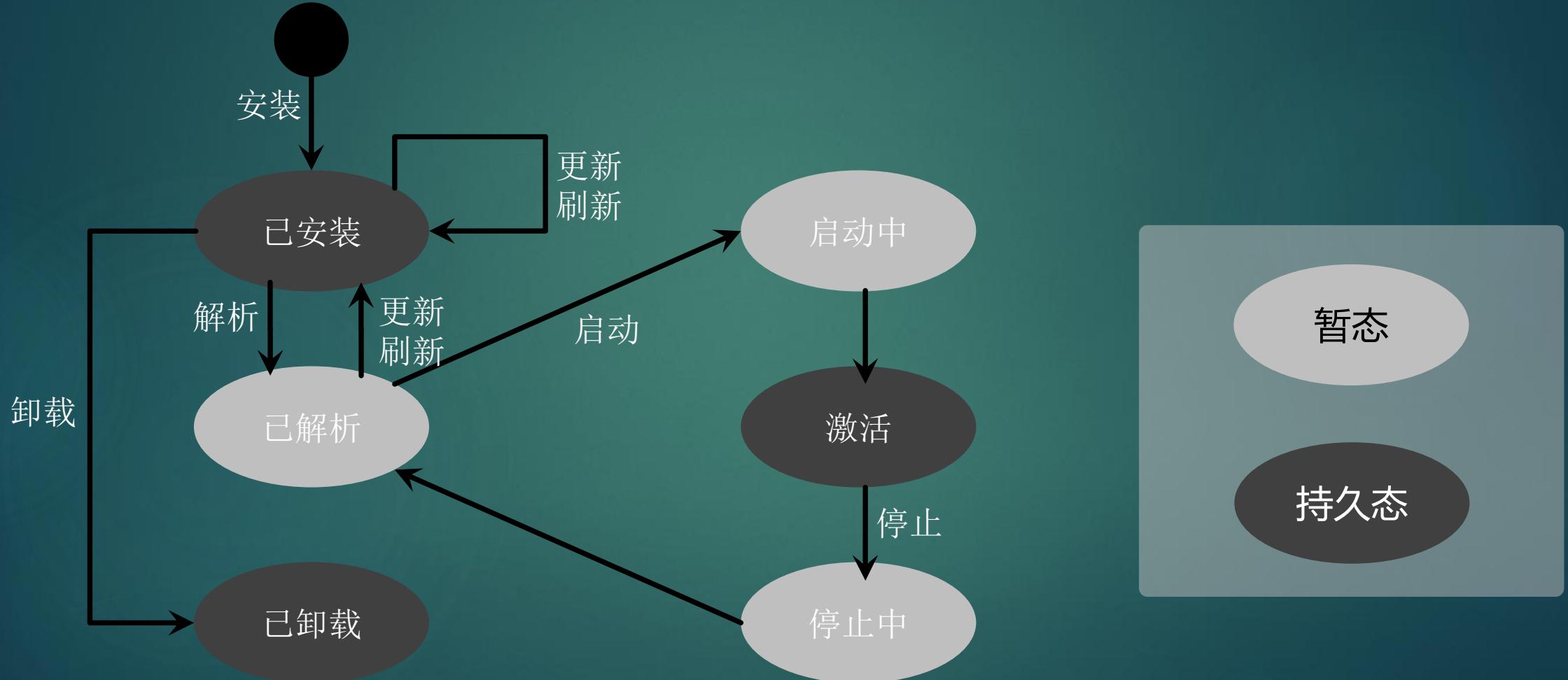
模块进行管理，包括安装，升级，

运行和卸载。

□ 另一亮点是OSGI允许在应用

运行时进行生命周期管理。

生命周期层：状态图



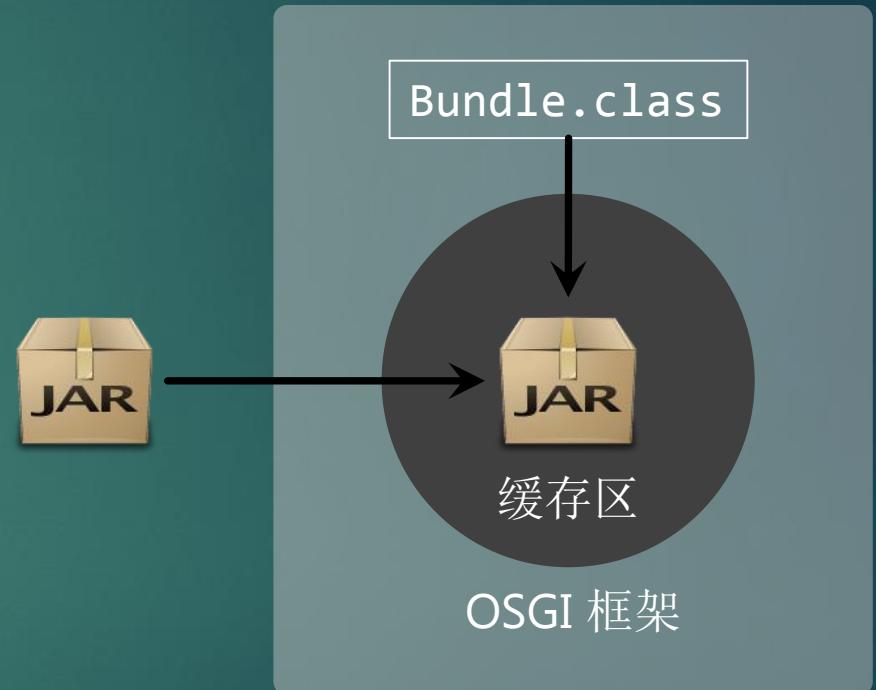
生命周期层：状态图—已安装

`BundleContext.installBundle(String location)`

拷贝 JAR 包到 OSGI 的缓存区

创建一个 `Bundle.class` 对象与之对应

`Bundle` 对象状态置为 `INSTALLED`



□ bundle 安装流程

生命周期层：API—Bundle接口

```
public interface Bundle{  
    int UNINSTALLED = 1;                                //卸载状态  
    int INSTALLED = 2;                                 //已安装状态  
    int RESOLVED = 4;                                //已解析状态  
    int STARTING = 8;                                //启动中  
    int STOPPING = 16;                               //停止中  
    int ACTIVE = 32;                                 //激活态  
  
    long getBundleId( );                            //获取bundle ID, 返回正整数ID  
    String getLocation( );                          //获取bundle在OSGI缓存区的地址  
    String getSymbolicName();                      //获取符号名  
    int getState( );                             //获取bundle当前所处状态  
    loadClass( String name );                    //使用bundle类加载器根据类名加载类  
    void start( );                                //启动bundle  
    void stop( );                                 //停止bundle  
    void uninstall( );                           //卸载bundle  
    void update( ); .....//这里只展示了目前关心的部分  
}
```

生命周期层：状态图—已安装

- 拷贝JAR包到OSGI缓存区，意味着原始JAR包已经不重要了。
就像软件安装完毕，安装包即可删除一样。
 - 缓存区会一直保存此bundle的JAR包，直到其被卸载。
 - 如果bundle已经安装过了，则不会重复安装，只返回对应Bundle对象。
 - 这是个持久状态，如果bundle已安装了，即便框架重启，它还是已安装的。
 - bundle及其生命周期由其Bundle对象管理。
-
- bundle安装完毕，即有一个Bundle对象代表它，OSGI 框架只认得这个逻辑bundle，此对象只能被框架创建。
 - Bundle对象有关于bundle的三个标识符：
 - ID：正整数，安装bundle时递增，在生命周期中不变更
 - location：对应OSGI 缓存区中此bundle JAR包的路径
 - SymbolicName：唯一标识bundle的元数据，在JAR包清单文件中由开发者指定
 - 你可以在Bundle接口中看到三个对应的函数。

生命周期层：状态图—已解析

- 当bundle要被启动时，OSGI会先让其依赖链上的bundles处于已解析状态。解析过程参见模块层之bundle模块的依赖。
- 已解析过的bundle不需要重新被解析，除非其退回到已安装状态。
- 这是OSGI 的一个隐式状态，即不像已安装状态那样是通过显示调用框架接口而得的。当框架发现一个已安装的bundle要被启动，就自动将bundle切换至此状态。
- bundle的解析是由框架自动执行的，但其实开发者也可以主动对bundle执行解析,后续讲到。

生命周期层：状态图—启动中



□ bundle启动流程

- 这也是OSGI 的一个隐式状态

- 调用 `Bundle.start()` 方法对 bundle 而言是个永久过程，即便框架重启，框架仍记得这个函数被调用过，从而会让bundle恢复至ACTIVE状态。除非调用 `Bundle.stop()` 或 `Bundle.uninstall()`方法去主动切换状态。

生命周期层：状态图—激活

Bundle对象状态为 STARTING



如果有激活器，则调用激活器的start()方法



Bundle对象状态置为 ACTIVE

□ bundle激活流程

MANIFEST.MF

Manifest-Version: 1.0

Built-By: some guy

Bundle-SymbolicName: xxx-toaster-provider

Bundle-Version: 1.1.0.Helium

Bundle-Activator: ToasterActivator

Export-Package: toaster_provider, ...

...

激活器是在清单文件里面指定的，相当于bundle的主类。

生命周期层：API—BundleActivator接口

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

OSGI框架的最外层接口，传入框架为此bundle创建的BundleContext上下文对象，让bundle与OSGI框架的交互成为可能。

```
public class ToasterActivator implements BundleActivator {  
    void start(BundleContext context){ ... }  
    void stop(BundleContext context){ ... }  
}
```

bundle 的 激 活 器 需 要 实 现
BundleActivator接口，在start方法里面可以做一些申请资源之类的初始化操作，在stop方法里面释放所有的资源。

生命周期层：API—BundleContext接口

```
public interface BundleContext {  
    Bundle getBundle( )                                //获取自身bundle  
    Bundle getBundle( long id )                         //根据bundle id 获取bundle  
    public Bundle[] getBundle( )                          //获取所以已安装的bundle  
    Bundle installBundle( String location )           //安装bundle  
    .....                                         //这里只展示了目前关心的部分  
}
```

- 每个有激活器的bundle都有与自己相关的BundleContext上下文，在激活器的start()和stop()方法间有效。该上下文为bundle独占，不可在bundle间共享，这样才足够安全。

生命周期层：状态图—激活

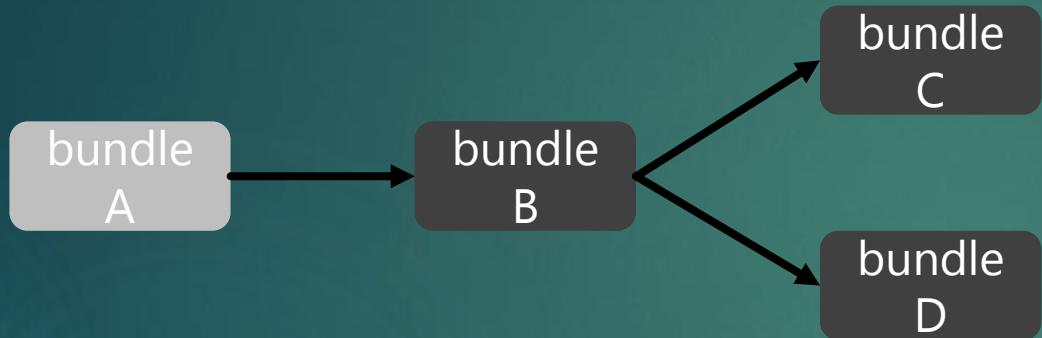
- bundle处于激活态，意味着它的类可以被加载并执行了。
- 激活器并不是必须的，只有当bundle需要做一些初始化操作或者要与OSGI框架交互时，它才需要有个激活器。
- 例如一个仅仅提供基础函数库的bundle就没必要有激活器。
- 一个激活器实例只在框架启动和停止bundle时使用一次，随后被丢弃。



激活器的作用

- 让bundle激活前做一些初始化
- 让bundle可以访问框架API

生命周期层：状态图—激活

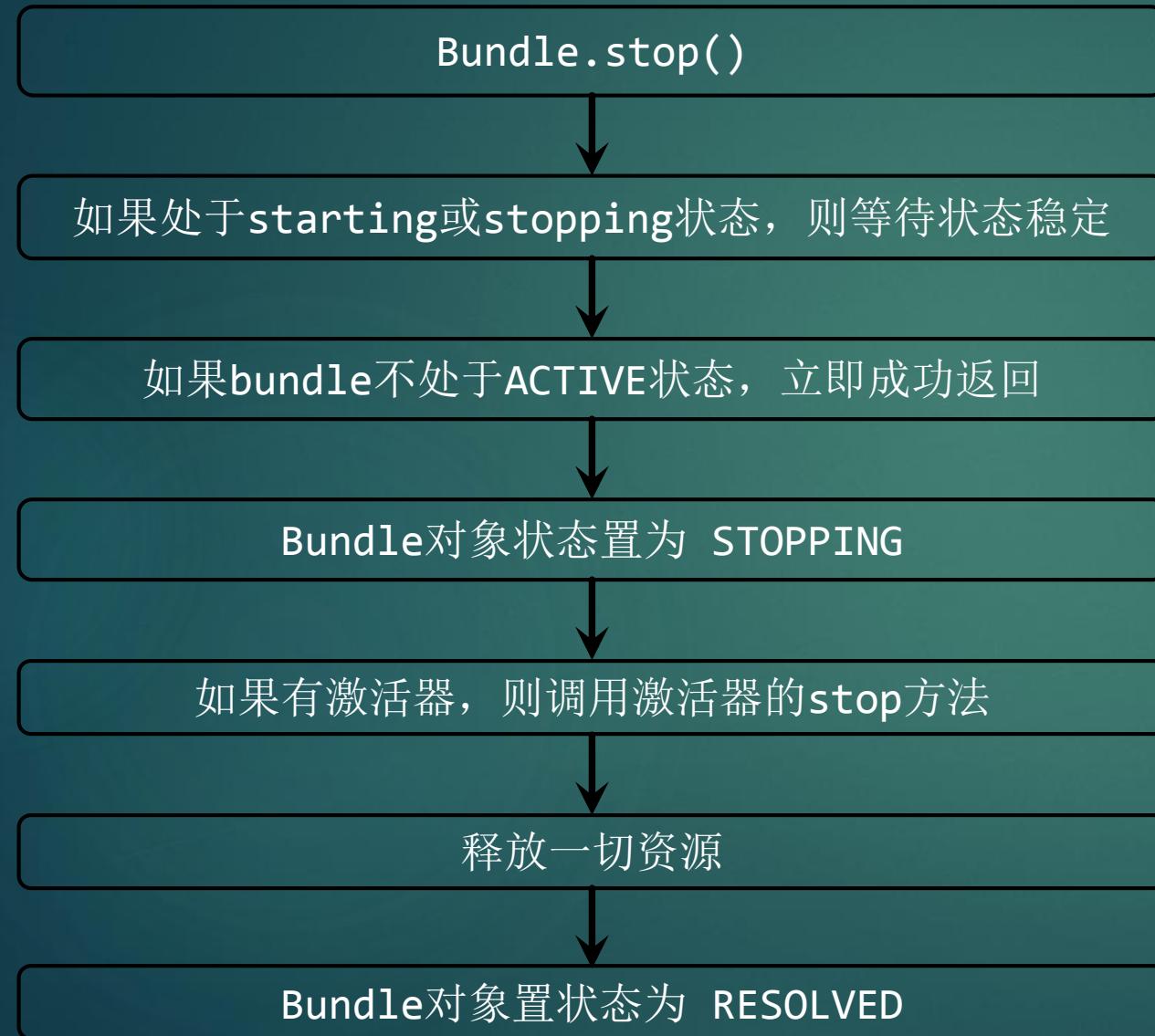


□ 我知道，一个bundle必须处于激活态它才可用，那我们考虑上面的场景。bundle A的依赖链解析成功，bundle A也已激活，它正在运行中。这时它需要使用bundle B导出的类，可bundle B并没有处于激活态，它的类不可用。这时会怎样？

启动优先级

1. 你的顾虑完全合理，实际的处理也并未超出你的预期，那就是抛出异常。
2. 可以大致知道，bundle会有启动优先级，框架会让被依赖的bundle先启动。但这目前只是猜测，此问题暂时留在这。

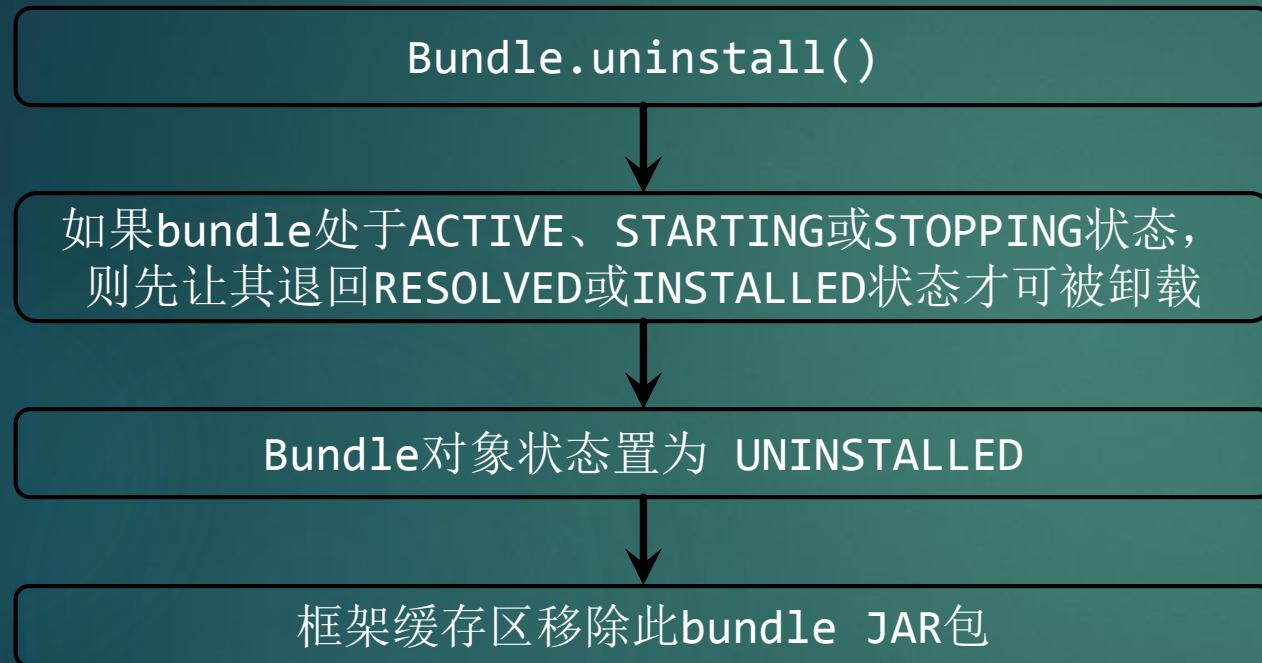
生命周期层：状态图—停止中



bundle停止，导出包不停止

1. 可能会让你惊讶，bundle B停止后，对bundle A并没有影响，bundle A可以照常使用bundle B的导出包。
2. 原因很简单，bundle B导出包的类被bundle A持有，还驻留在内存。
3. 如果你不喜欢这样，记得导出包应只有接口而没有实现。
4. 或者对bundle B执行刷新(后面讲)。

生命周期层：状态图—已卸载



bundle停止，导出包不停止

1. 卸载一个处于激活态的bundle，还是先走 stop 流程，所以其导出包仍然可以被其他bundle使用，原理是一模一样的。

□ bundle卸载流程

生命周期层：状态图—更新



bundle更新，导出包不更新

1. 更新一个处于激活态的 `bundle`，还是先走 `stop` 流程，所以其导出包仍然可以被其他 `bundle` 使用，原理是一模一样的。
2. 即便新的 `bundle B` 回到 `ACTIVE` 状态，但 `bundle A` 使用的仍是其旧的导出包，仿佛什么事都没发生过似的。

生命周期层：状态图—总述

- bundle的六个状态中，三个状态是持久的，另三个状态是暂时的。所谓持久和暂时，是相对框架会话而言的，框架从其启动到停止为一次会话。

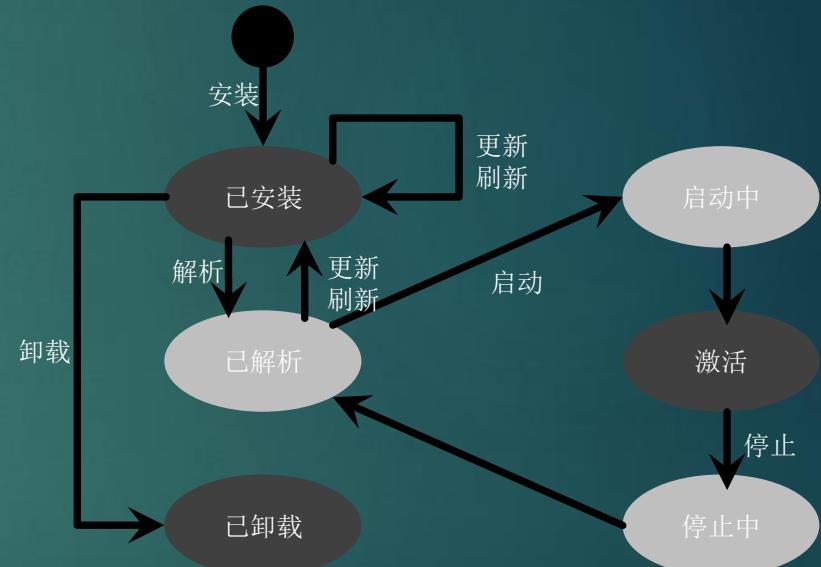
持久态

已安装和已卸载很好理解。对于激活态，不论框架重启多少遍，它都会按照状态机的流程去尝试恢复bundle所标记的激活态。

暂态

bundle为达到持久态所必须经历的中间态，框架不会在不同会话间去记忆这种中间态。

- bundle的生命周期要由其他bundle去操控，即bundle不要自己调用start、stop之类的方法，否则一不小心就异常。



生命周期层：API — 总述



□ BundleActivator：激活器，用于bundle启动时进行初始化，是bundle访问框架API 的桥梁。



□ BundleContext：上下文，bundle独自占有的对象，提供了框架API。



□ Bundle：每个已安装的bundle都有一个Bundle对象与之对应，逻辑上代表它，用于对bundle进行管理，尤其是生命周期管理。

OSGI 标准API

□ 承诺生命周期管理的API，而不提供管理工具，这样可以有无数种管理bundle生命周期的方式。我们使用的shell只是方式之一，可以开发自己的其他任何方式。

系统bundle：不应该隐瞒你，其实还有个所谓系统bundle的对象，对于OSGI框架，其ID为0，框架运行时一直存在。

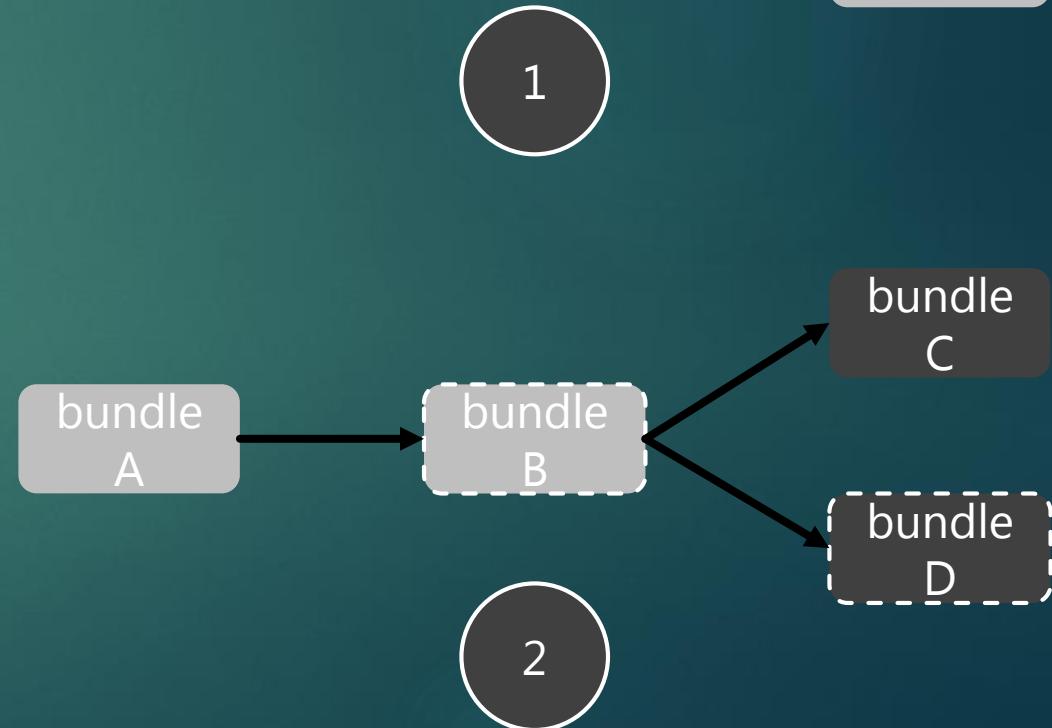
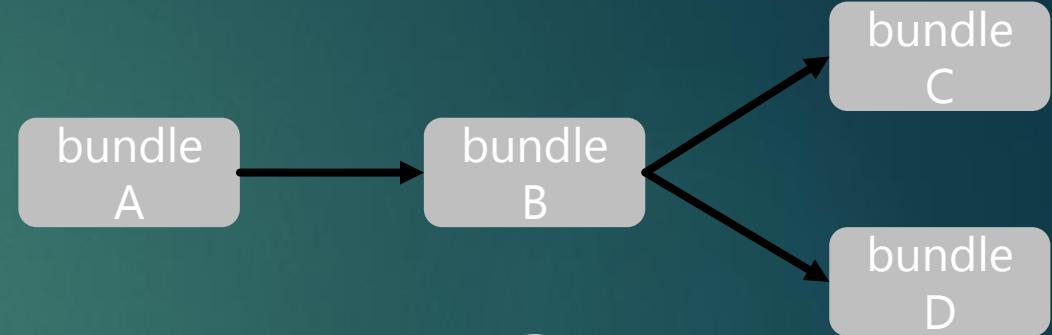
它有着和普通bundle一样的生命周期，可以一样的操纵它，但要小心，因为你调用其bundle.stop()方法时，会先停止所有其他bundle，再关闭框架。

生命周期层：运行时的动态性



记得OSGI 承诺过，允许在应用运行时对其组成bundle进行生命周期管理，如升级、卸载，但不知道它是怎么处理的？

- 是这样的，如图1，几个bundle都在激活态运行，而且运行的很正常。它们之间有一些依赖关系。
- 但后来事情发生了变化，bundle B被更新了，bundle C被停止了，bundle D被卸载了，bundle A保持不变，如图2。完全无法预料，接下来会发生什么？

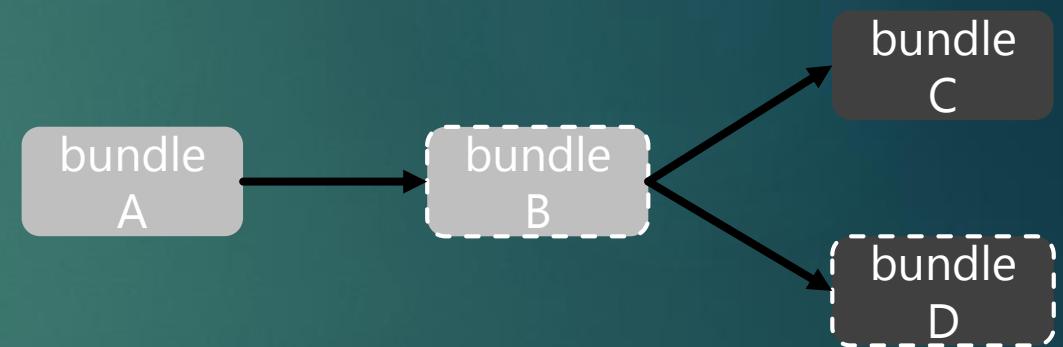


生命周期层：运行时的动态性

1. 首先不论是刷新、停止或是卸载，它们都先走stop流程。所以即便bundle已经发生变化，但依赖它的bundle并不感知(除非监听这些变化)，照常使用其旧的导出包。所以bundle A还是可以继续运行的。

- A □ 这里问题的根源在于，已解析的bundle不会重新解析，即便它们已经发生变化，bundle A 仍依赖的是旧的解析结果。

2. 要让已经发生改变的bundle产生作用，需要对这些bundle重新解析，或者更简单的调用包管理器的刷新操作 `PackageAdmin.refreshPackages(...)`。



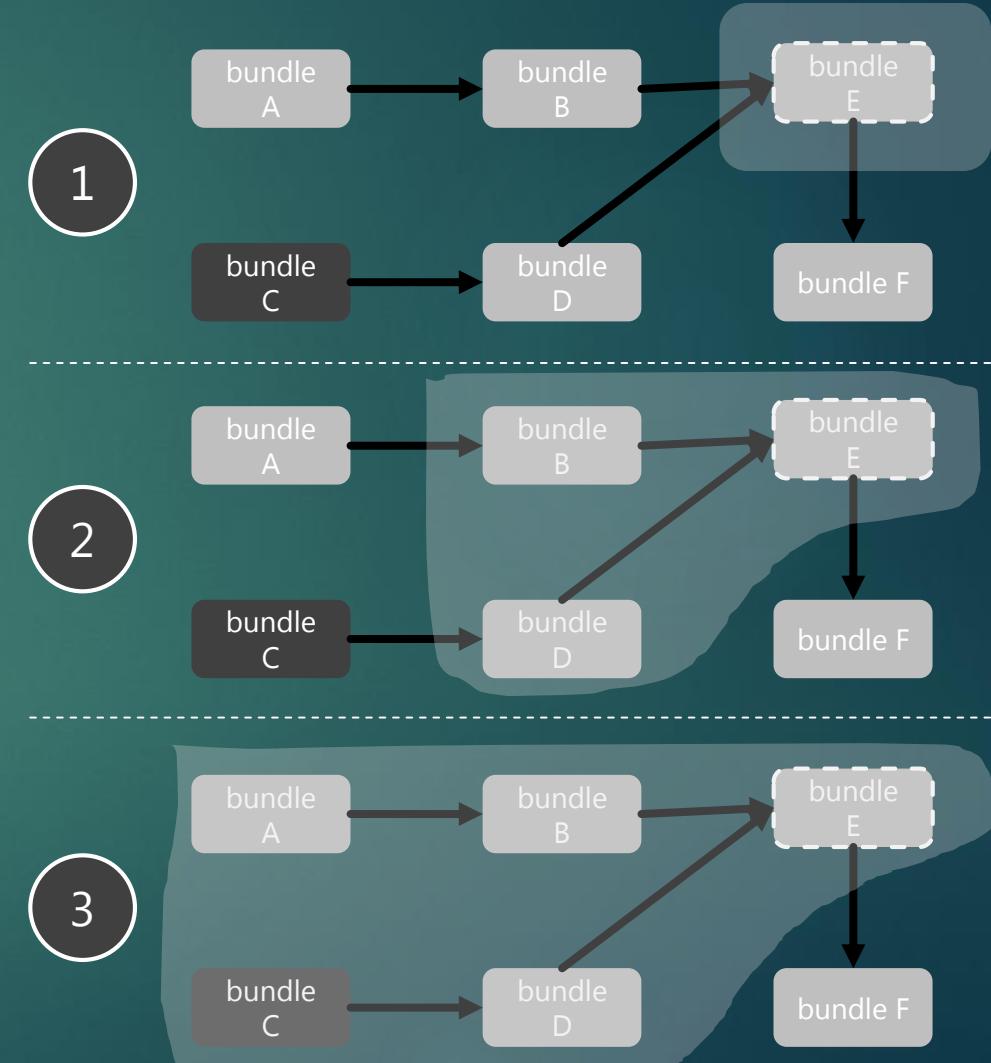
生命周期层：API—PackageAdmin接口

```
public interface PackageAdmin {  
    ExportedPackage[] getExportedPackages( ... );//重载方法          //获取bundle导出包  
    void refreshPackages(Bundle[] bundles);                      //刷新bundle  
    boolean resolveBundles(Bundle[] bundles);                    //解析指定bundle  
    RequiredBundle[] getRequiredBundles(String symbolicName);  
    Bundle[] getBundles(String symbolicName, String versionRange);  
    Bundle getBundle(Class clazz);                                //根据类名获取bundle  
}
```

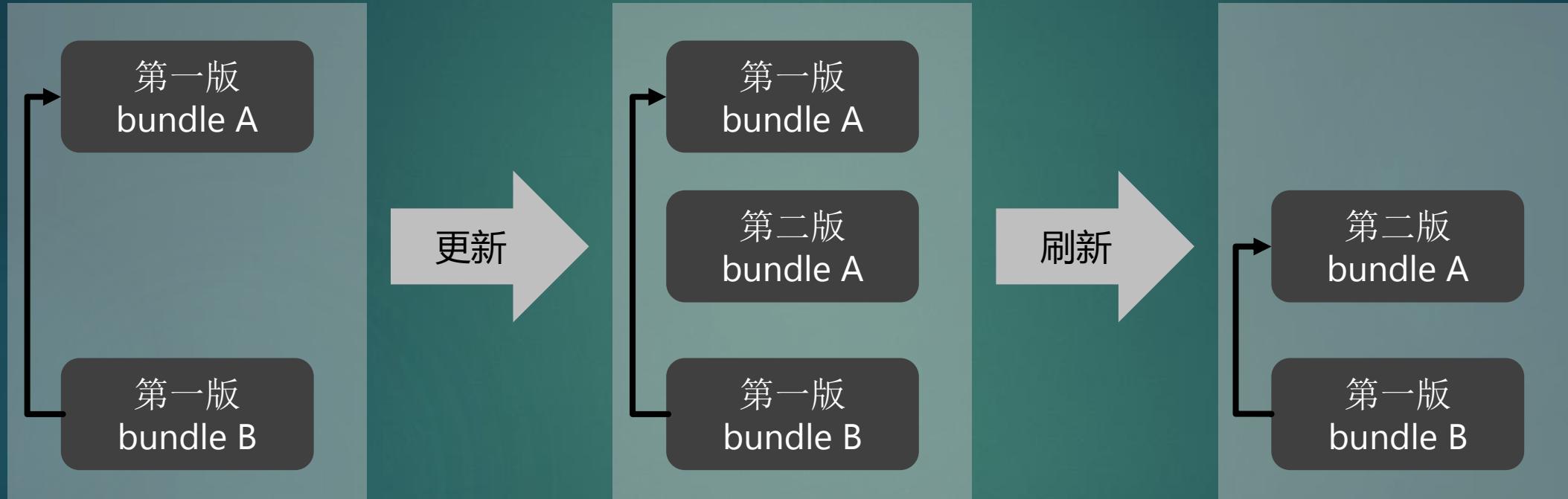
- 包管理器，bundle解析，bundle刷新都是在这里被调用的

生命周期层：运行时的动态性

1. 调用 `PackageAdmin.refreshPackages(...)`，刷新指定的 bundles，如参数为null，则刷新所有发生变更的bundles，将它们加入一个待刷新的数组中。右图1。
2. 从指定的或发生变更的bundles开始，计算所有受到影响的 bundles，将它们添加到待刷新数组，直至没有其他bundle 依赖于数组中的bundles。右图2,3。
3. 数组中的每个bundle依次退回至INSTALLED状态，全都变成 未解析的。数组中每个被卸载的bundle从数组移除。
4. 对数组中的每个bundle，如果其原来处于激活态，则经过解析后让其恢复至激活态。如果依赖无法满足则抛出异常。
5. 所有这些都在一个独立的线程里完成。



生命周期层：运行时的动态性



□ bundle更新完整过程

生命周期层：模块层与生命周期层

- 对开发者而言，模块层依赖于元数据，生命周期层依赖于API。
- 生命周期层需要关注的元数据：Bundle-Activator

一个运行于OSGI 之上的应用，其所有bundle都没有激活器，都不使用OSGI 提供的API，所有的bundle仅仅使用了模块层定义的元数据，那也是可以的。其实这和传统的java程序开发没有区别，使用标准方式开发的java程序，也可以简单修改元数据而迁移进OSGI 框架。在这种情况下，OSGI 的存在价值在哪？

首先，在模块层，一个bundle仅仅导出包对外可见，OSGI 也附带做好了依赖解析的工作，由此而强化了java模块化的概念，这是标准java程序所缺乏的。

其次，在生命周期层，OSGI 允许以bundle为单位，在运行时对应用进行部署管理，这样提供了足够的灵活性。所以，即便在OSGI 框架之上的应用没有调用OSGI API，也享受到了OSGI 框架带来的好处。

生命周期层



OSGI bundle 变更会导致一批bundles产生大动荡，我并不认为这很优雅很灵活！



确实，但先别忙着下定论，OSG 提供的不仅仅如此！

欲知更多，请看后续 OSGI 之 **服务层** >>>

