ORACLE®

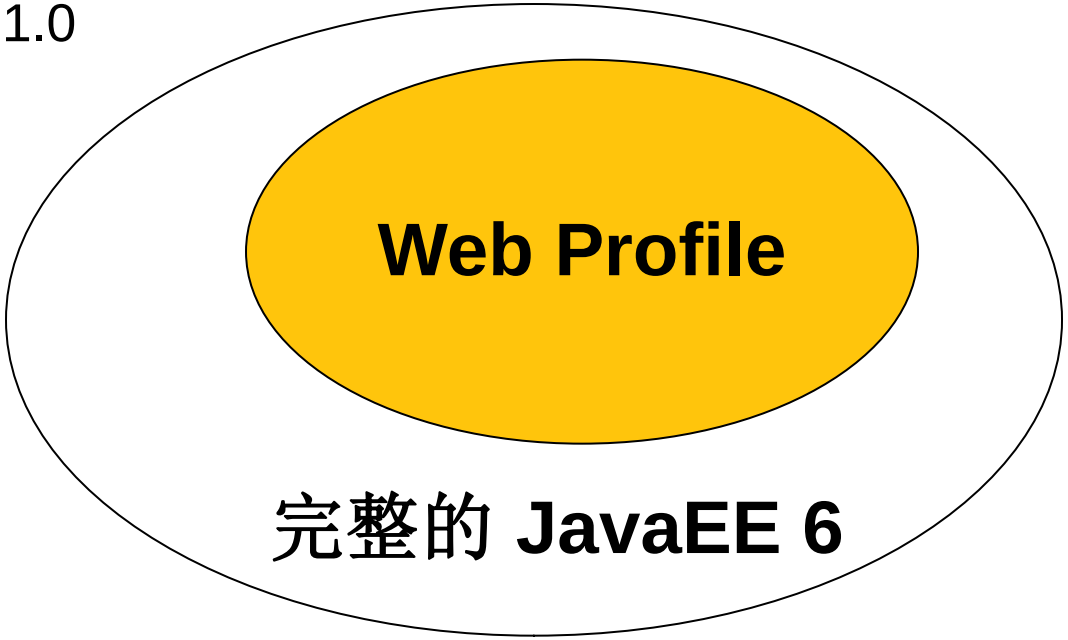
JavaEE 6
Servlet 3.0 中的新特性

议题

- 易于开发和部署
- 模块化 web.xml
- 动态配置
- 异步 Servlet

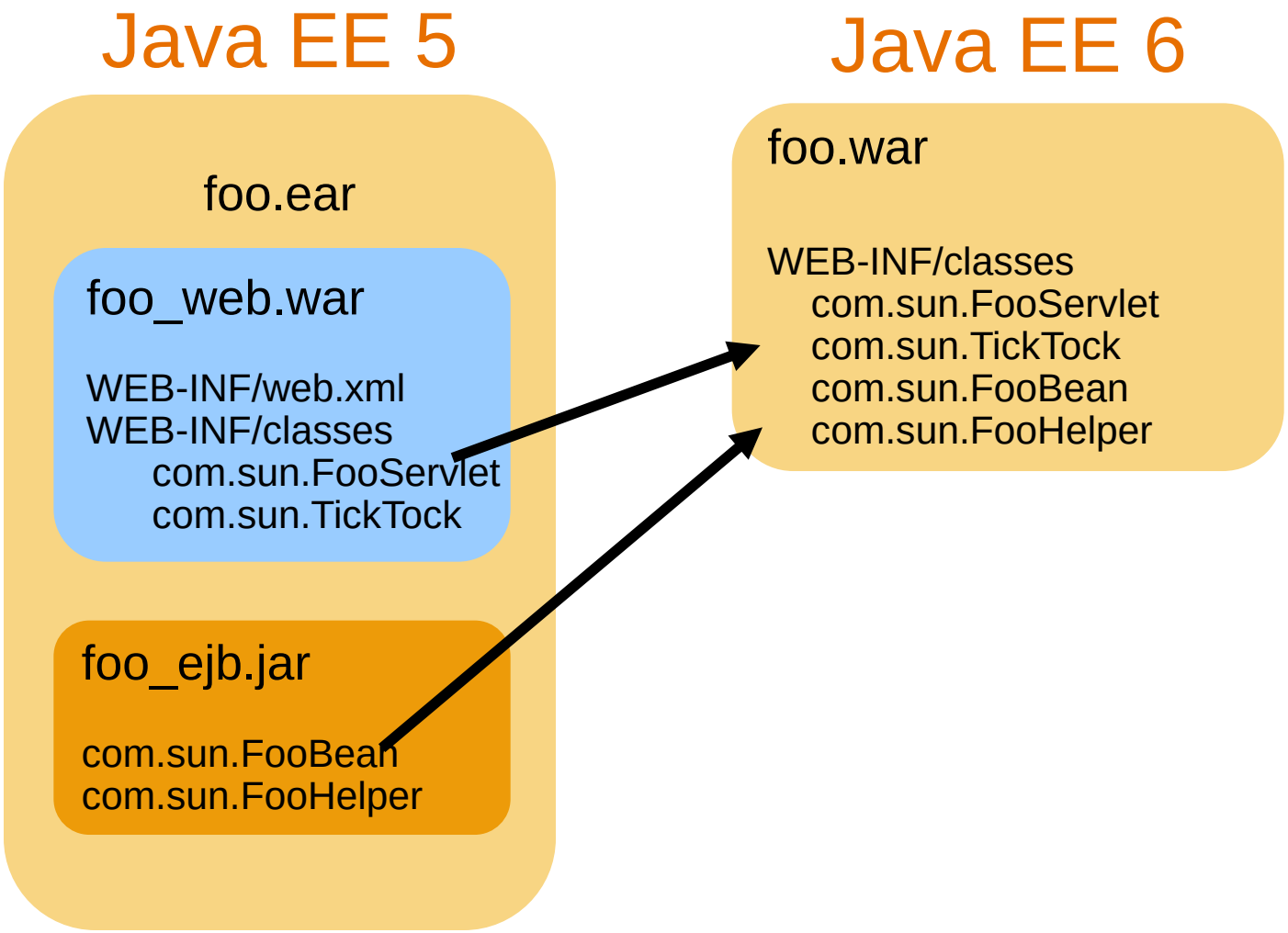
Web Profile

- 用于 Web 应用程序的功能全面的中型 profile
- web profile 中的各种技术
  - Servlet 3.0、JSP 2.1、对其他语言的调试支持、EL 1.2、JSTL 1.2、JSF 2.0
  - EJB Lite 3.1、JTA 1.1、JPA 2.0、通用批注
  - Bean validation 1.0



易于开发和部署

简化的打包


Java EE 5
foo.ear
foo_web.war
WEB-INF/web.xml
WEB-INF/classes
com.sun.FooServlet
com.sun.TickTock
foo_ejb.jar
com.sun.FooBean
com.sun.FooHelper
Java EE 6
foo.war
WEB-INF/classes
com.sun.FooServlet
com.sun.TickTock
com.sun.FooBean
com.sun.FooHelper

Servlet — 下一代

- Servlet 3.0 — 易于开发
  - 主要关注点
- 增强了 API 以便使用 SE 5 中新的语言特性
  - 例如：批注、泛型(Generics)
  - 在上次 JavaEE 5 中漏掉的 Servlet
- 部署描述批注
  - 现在 web.xml 为可选
- 用于保证 API 类型安全的泛型(Generics)
  - 不影响后向兼容性
- 惯例优先原则(Convention over)
  - 更好的默认值
  - 例外配置

Servlet 2.x

```
public class HelloServlet extends HTTPServlet {
        public void doGet(HttpServletRequest req
                , HttpServletResponse res) throws Exception {
        …
        }
}
<web-app>
        <servlet>
                <servlet-name>HelloServlet</servlet-name>
                <servlet-class>myservlet.HelloServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>HelloServlet</servlet-name>
                <url-pattern>/hello</url-pattern>
        </servlet-mapping>
        . . .
</web-app>
```

Servlet 3.0 — 示例 1

```
@WebServlet(name="HelloServlet", urlPatterns="/hello")
public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest req
        , HttpServletResponse res)  throws Exception {
    . . .
}
```

无 web.xml

```
//name="HelloServlet", urlPatterns="/hello"
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest req
        , HttpServletResponse res) throws Exception {
    . . .
}
```

@WebServlet

- 用于**指定** servlet
- 批注至少必须有 urlPattern
  - urlPattern 属性是默认属性
    - 即 value()
- 所有其他字段都有*合理的*默认值
  - 例如，servlet 的名称默认为 servlet 的全称类名
- 类的其余内容与之前相同
  - 必须扩展 HttpServlet 类
  - 重写 doXXX() 来处理行为
- @WebListener、@WebFilter 的工作方式相同

Servlet 3.0 — 示例 3

```
@WebServlet(
  urlPatterns="/hello", loadOnStartup=1
, initParams={
          @InitParam(name="db", value="jdbc:derby...")
        , @InitParam(name="dbuser", value="fred")
        , @InitParam(name="dbpasswd", value="fred")
    }
)
public class HelloServlet extends HttpServlet {
public void init(ServletConfig config) throws ServletException {
    String jdbc = config.getInitParameter("db");
    …
}
…
```

使用批注

- 用于部署 servlet、过滤器、监听器的批注
  - @WebServlet — 向 web.xml 中添加 servlet 项
  - @WebFilter — 添加过滤器项
  - @WebListener — 添加监听器项
  - @WebInitParam — 添加参数项
  - @MultiPartConfig —表示 servlet 要求使用 mime/multipart 数据格式
- 在部署时使用 web.xml 覆盖批注值

Servlet 3.0 — 示例 2

```
@WebFilter(urlPatterns="/hello"
, dispatcherTypes={
            DispatcherType.FORWARD}
)
public class HelloServletFilter
     implements Filter {

public void doFilter(HttpServletRequest req
          , HttpServletResponse res) {
     . . .
}
```

Multipart 支持

- Multipart 格式用于在 HTTP 流中分割传输内容
  - 通常是表单数据、非 ASCII 数据、二进制数据等

```
<form action=”/upload”>
Name: <input name=”filename” type=”text”/>
File to upload: <input name=”myfile” type=”file”/>
<input type=”submit” value=”Upload”>
</form>
POST /upload HTTP/1.0
Content-Type: multipart/form-data; boundary=----ABCDEFGHIJK
…
----ABCDEFGHIJK
Content-Disposition: form-data; name=”filename”

uploaded_file.txt
----ABCDEFGHIJK
Content-Disposition: form-data; name=”myfile”; filename=”a_text_file.txt”

…
```

Multipart 支持

- Servlet 目前原生支持 multipart 数据
  - 以前必须借助于第三方库
- 获得 MIME 内容的新方法
  - HttpServletRequest.getParts() – Iterable<Part>
  - HttpServletRequest.getPart(String name) – String
- 类似功能
  - HttpServletRequest.getParameter(“name”)
  - HttpServletRequest.getPart(“name”)
    - 比前者更通用的形式

@MultipartConfig 批注

- 帮助 servlet 容器确定哪个 servlet 将处理 multipart 数据
- 配置容器处理数据的方式
  - 临时文件的位置
  - 最大文件大小

```
@WebServlet(“/hello”) @MultipartConfig(location=”/tmp”)
public class HelloServlet extends HttpServlet {

public void doPost(HttpServletRequest req, HttpServletResponse res) {
      Part part = req.getPart(“myfile”);
      InputStream is = part.getInputStream();
      //Read in file
      . . .
}
```

模块化 web.xml

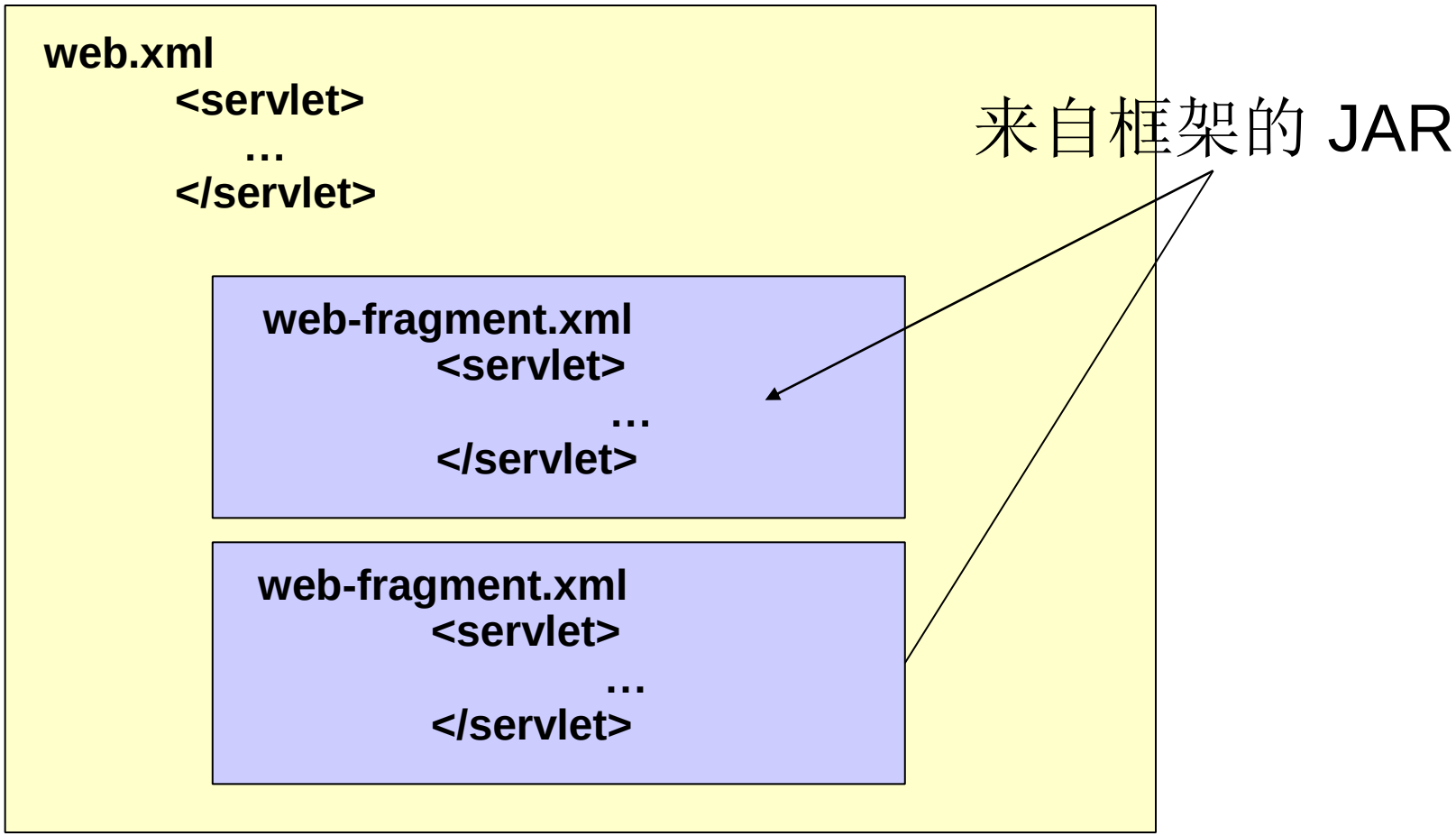
最新发展 — web.xml

- 多种非常高效的 web 框架
- 框架的使用需要（可能是复杂的）配置
  - 如果使用多种框架则难以管理 — 复杂性增加
  - 需要知道各框架之间的整合方式
- 例如
  - 声明控制器 servlet 及其参数
  - 处理请求前后所使用的过滤器
  - 声明监听器以便管理应用程序生命周期的各点
- 很多这种配置在应用程序间通用

模块化 web.xml

- 允许在没有框架配置的情况下使用框架
  - 为您的应用程序保留 web.xml
  - 将框架配置工作放在框架这边
- 允许框架将它们的 web.xml 存储在自己的 JAR 中
  - 在应用程序启动期间合并它们
- web.xml 分段的概念

web.xml 与 web-fragment.xml



web-fragment.xml

- 用于指定整个 web.xml 的一部分
- 除根元素外与 web.xml 相同
  - <web-fragment>
- 位于框架的 JAR 中
  - META-INF/web-fragment.xml
  - JAR 文件必须位于 WEB-INF/lib 中

web-fragment.xml 示例

```
<web-fragment>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    . . .
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup> 1 </load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
    . . .
</web-fragment>
```

web-fragment.xml 示例

jsf_framework.jar

META-INF/web-fragment.xml

mywebapp.war

WEB-INF/lib/jsf_framework.jar

资源共享

- 静态页面和 JSP 页面，资源不再局限于应用程序的 DOCROOT
  - 允许框架捆绑自己的资源
- 资源位于框架的 JAR 文件中
  - META-INF/resources
  - ServletContext.getResourceAsStream() 将知道如何解析资源位置
- 应用程序 DOCROOT 中的资源优先于框架 JAR 文件中的资源
  - 如果存在命名空间冲突
- “/” 对应 DOCROOT 或 META-INF/resources

资源共享示例

mywebapp.war

/index.jsp

/WEB-INF/lib/framework.jar!

/META-INF/web-fragment.xml

/META-INF/resources/index.jsp

/META-INF/resources/shared.jsp

http://server.com/mywebapp/shared.jsp

mywebapp.war!/WEB-INF/lib/framework.jar!

/META-INF/resources/shared.jsp

http://server.com/mywebapp/index.jsp

mywebapp.war!/index.jsp

动态配置与按需加载

动态配置

- 以编程方式添加 servlet、监听器和过滤器
  - 例如，自动添加 Strut 的 ActionServlet
- 只能在下列各项中执行配置
  - ServletContextListener.contextInitialized()
    - 由应用程序实现
    - 应用程序启动时调用
  - ServletContainerInitializer.onStartup()
    - 稍后详细介绍
  - 使用返回的 Registration 对象进一步定制
    - 例如，映射、初始化参数、servlet 名称等

web.xml 配置

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
        org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>
            /WEB-INF/struts-config.xml
        </param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

编程式配置

```
@WebListener(“Auto configure Struts”)
public class MyListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent evt) {
        ServletContext ctx = evt.getServletContext();
        //Register the servlet
        ServletRegistration.Dynamic servReg = ctx.addServlet(
                “action”, “org.apache.struts.action.ActionServlet”);
        //Set init parameter
        servReg.setInitParameter(“config”
                , “/WEB-INF/struts-config.xml”);
        servReg.setLoadOnStartup(2);
        //Add mappings
        servReg.addMapping(“*.do”);
        //Annotate with @MultipartConfig
        servReg.setMultipartConfig(
                new MultipartConfigElement(“/tmp”));
    }
}
```

动态配置的局限

- 注册仅发生在应用程序启动时
  - 之后不能更改
- 无法以编程方式删除 servlet
  - 无法从 web.xml 删除项
- 只能在其上下文或上下文分支中安装 servlet/过滤器/监听器
  - 例如 /myapp、/myapp/invoice
- 无法共享全局状态
  - 每个“分布式”应用程序一个实例

按需加载

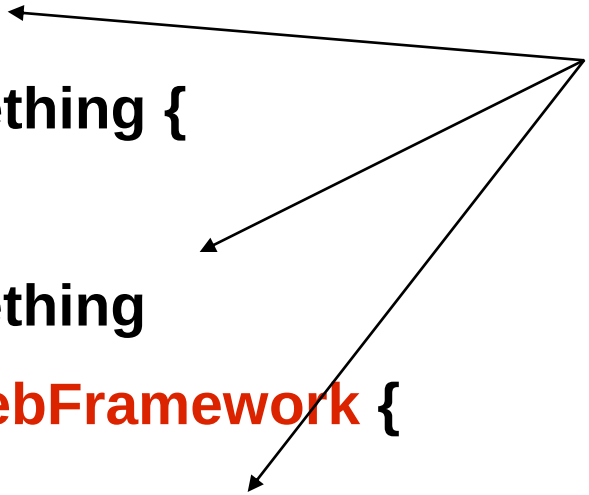
- 按需加载库和框架
  - JSF、web 服务、第三方框架
- 应用程序启动时
  - Web 容器找到使用的框架
  - 调用 ServletContainerInitializer.onStartup() 来初始化框架/库
- 功能类似于 ServletContextListener.contextInitialized()
  - ServletContextListener.contextInitialized() 由应用程序开发人员实现
  - ServletContainerInitializer.onStartup() 由框架开发人员实现

ServletContainerInitializer

- 框架实现 ServletContainerInitializer
  - 应用程序启动时调用 onStartup()
  - 负责初始化框架 — 如果有
    - 例如，注册控制器 servlet、连接到数据库等
- 通过 @HandlesTypes 实现批注
  - 将 initializer 与它的批注、类或接口集相关联
  - 例如 @HandlesTypes({Fred.class})
- 在 META-INF/services/ 中注册
  - 在运行时由 JavaEE 发现
- Web 容器扫描 web 应用程序来寻找 @HandlesTypes
  - 找到时调用 initializer

按需加载示例 — 1

```
@MyWebFramework
public class DoSomething {


public class DoSomething
        extends MyWebFramework {


public class DoSomethng
        implements MyWebFramework {
```

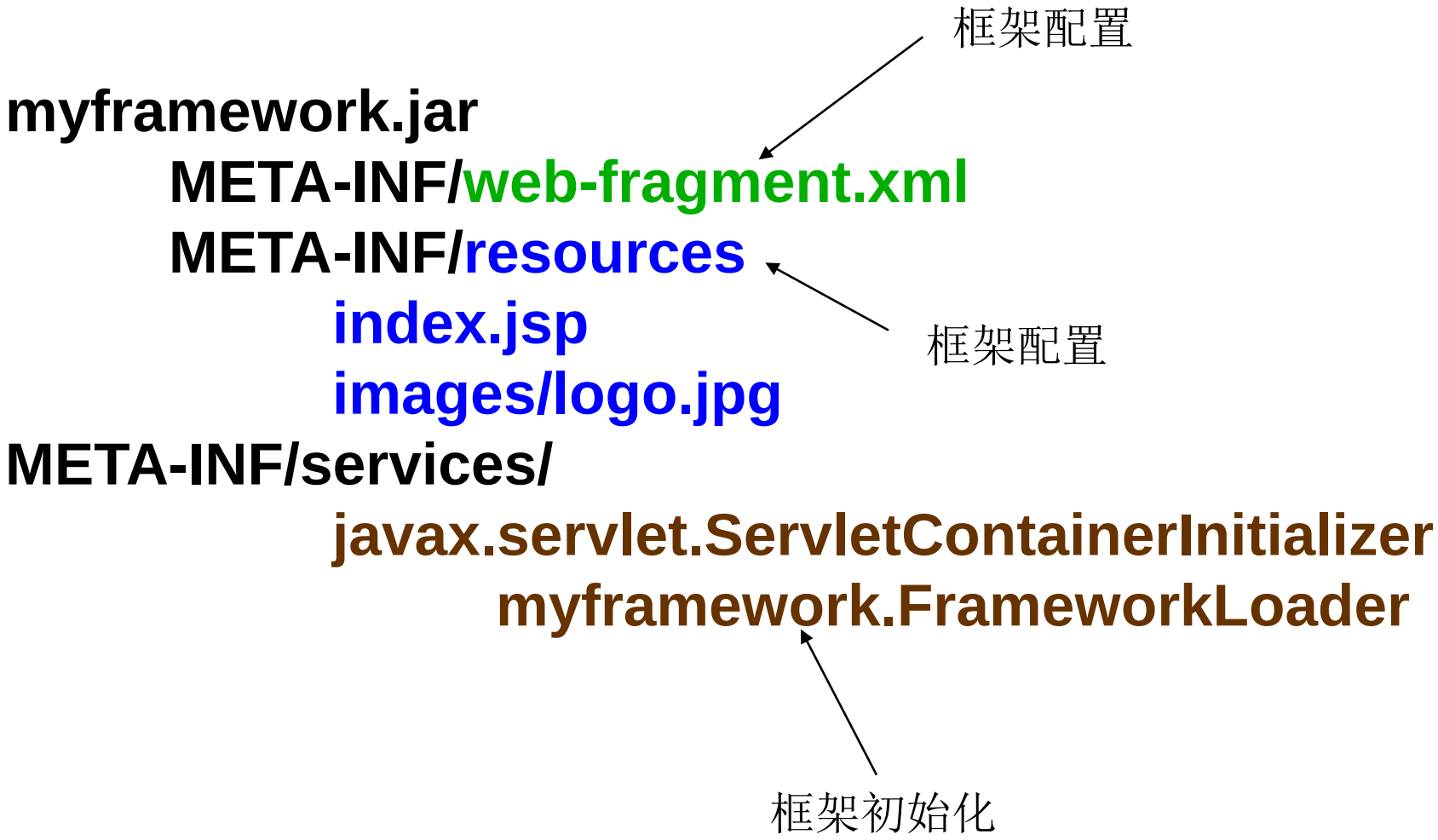


按需加载示例 — 2

myframework.jar
META-INF/services/
javax.servlet.ServletContainerInitializer
myframework.FrameworkLoader

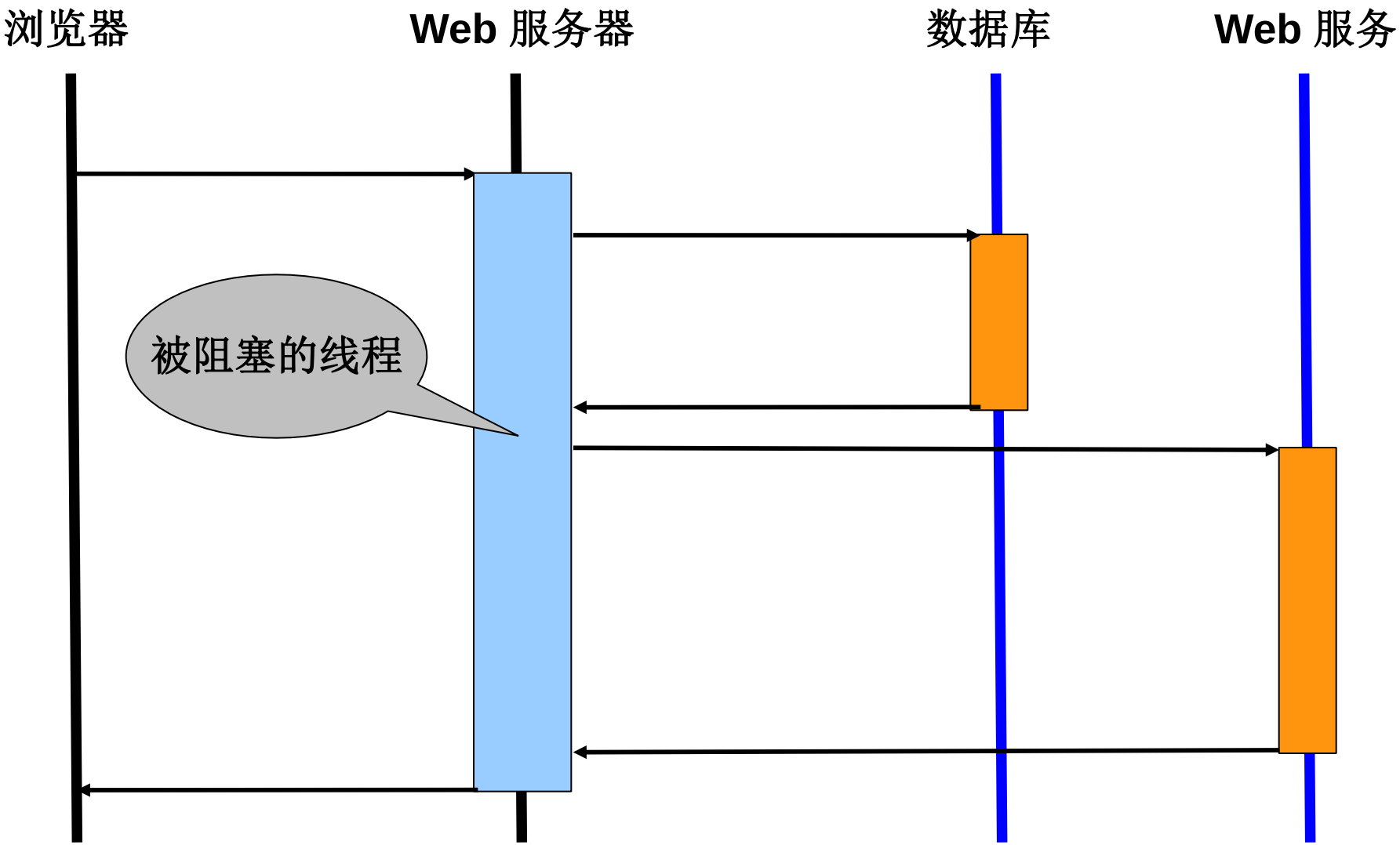
```
@HandlesTypes(MyWebFramework.class)
public class FrameworkLoader implements ServletContainerInitializer {

    public void onStartup(Set<Class<?>> c, ServletContext ctx) {
        //Load framework controller
        ctx.addServlet("SuperFrameworkController"
            , "myframework.controller.FrameworkController");
        //Other processing
        . . .
    }
}
```

使用框架时解析……


框架配置
myframework.jar
META-INF/web-fragment.xml
META-INF/resources
index.jsp
images/logo.jpg
框架配置
META-INF/services/
javax.servlet.ServletContainerInitializer
myframework.FrameworkLoader
框架初始化

异步 Servlet

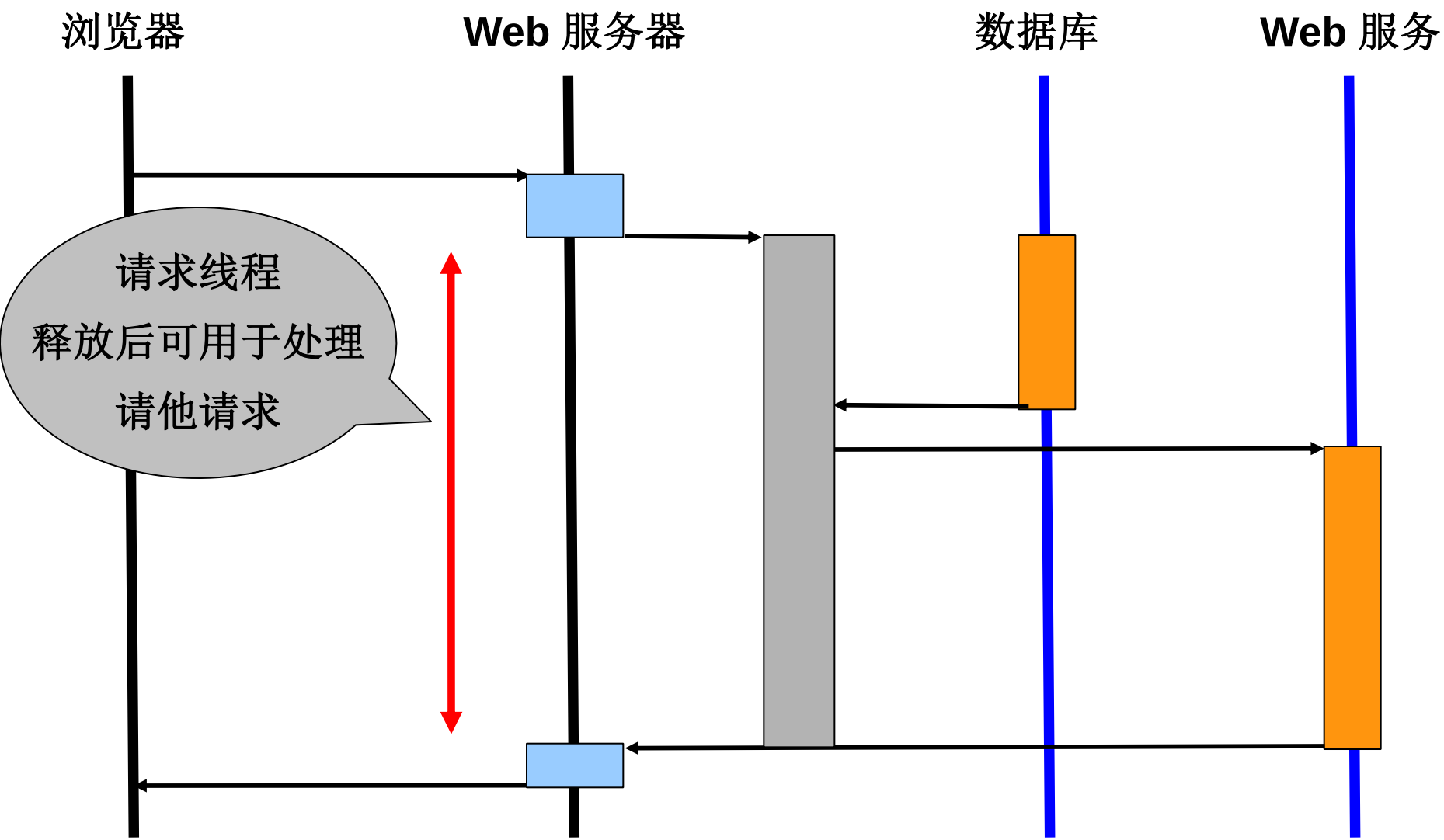
等待反应缓慢的资源



为什么要使用异步 Servlet?

- 非常适用于以下情况的 web 应用程序
  - 长处理时间或伪长处理时间
  - 等待资源释放 — 如数据库连接
  - 等待事件发生 — 如聊天消息
  - 等待缓慢服务的响应 — 如 web 服务
- 释放当前请求处理以便处理其他请求
  - 为 web 容器提供了更好的可伸缩性
- 浏览器将仍然显示为等待
  - 用户体验没有改变
- 不要与异步 I/O 混淆
  - 目前的 Servlet 3.0 不支持

异步 Servlet



AsyncContext

- 调用 HttpServletRequest.startAsync() 将 servlet 置于异步模式
  - 流将不在从 Servlet.service() 返回时提交
  - 返回 AsyncContext 对象
- AsyncContext.start() 启动请求
  - 将其指派给工作线程
- AsyncContext.dispatch()
  - 通知工作已经完成
  - 将控制传递回 web 容器

异步 Servlet 示例 — 1

@WebServlet(urlPatterns=”/hello”, asyncSupported=true)

```
public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest req
        , HttpServletResponse res) throws Exception {
    //Put the request in async mode
    AsyncContext asyncCtx = req.startAsync();

      //Create an instance of handler
      SlowResourceHandler handler =
            new SlowResourceHandler(asyncCtx);

     //Starts the process
     asyncCtx.start(handler);
     //Request will not commit on return
    }
```

异步 Servlet 示例 — 2

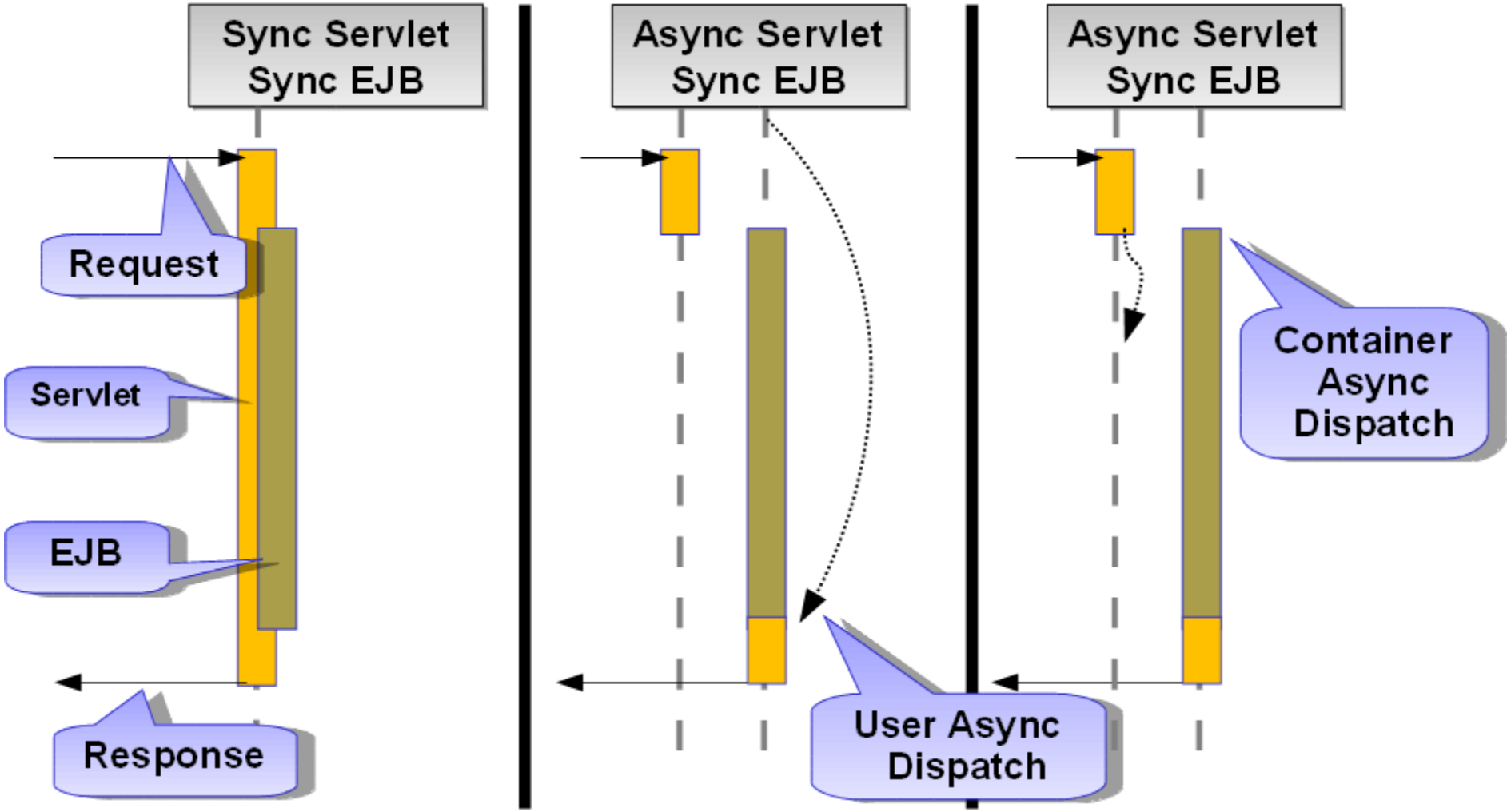
```
public class SlowResourceHandler implements Runnable {

    private AsyncContext asyncCtx;
    public SlowResourceHandler(AsyncContext c) {
        asyncCtx = c;
    }

    public void run() {
        //Perform some long running task
        . . .
        //Dispatches to the response
        //Completes the request
        asyncCtx.dispatch(“displayResult.jsp”);
    }
}
```

案例研究 — 结账

购物车结账

- 结账操作成本高昂、耗时漫长
  - 创建订单、处理商品（例如，特价商品）、信用卡验证和支付、更新客户帐户、制作交货单等
- 在完成所有这些独立的操作之前，请求线程将被阻塞
- 可选择分流到异步 servlet
  - 请求线程将被释放
  - 在结账完成时处理响应
  - 允许独立于处理线程限制请求线程

结账 — 3 种实现


Sync Servlet
Sync EJB
Request
Servlet
EJB
Response
Async Servlet
Sync EJB
User Async
Dispatch
Async Servlet
Sync EJB
Container
Async
Dispatch

异步 EJB

- EJB 3.1 引入了异步 EJB
  - 在类或方法上使用 @Asynchronous 批注
  - 以 void 作为返回类型，实现即发即弃
  - 以 Future<?> 作为返回类型，实现稍后获取结果
- EJB 的执行由容器调度
- 可将 EJB 打包在 WAR 文件中
  - 简化的打包

CheckOutServlet

```
@WebServlet("/checkout", asyncSupported=true)
@ServletConstraint(
   value=@HttpConstraint(transportGuarantee=CONFIDENTIAL)
   httpMethodConstraints={@HttpMethodConstraint(
      value="POST", rolesAllowed={"authenticate"}))

public class CheckOutServlet extends HttpServlet {
   ...
   @EJB CheckOutBean checkoutBean;
   public void doPost(... request, ... response) {
      //Perform some checking
      ...
      AsyncContext ctx = request.startAsync();
      checkoutBean.processAsync(ctx);
      //Request thread freed on exit
   }
```

CheckOutBean

```
@Stateless
public class CheckOutBean {
   @TransactionAttribute(REQUIRED)
   @Asynchronous
   public void processAsync(AsyncContext ctx) {
      try {
         //Process checkout
         ...
         ctx.dispatch("/checkout-success");
      } catch (Exception e) {
         sessionContext.rollbackOnly();
         ...
         ctx.dispatch("/checkout-failed");
      }
   }
```

总结

总结

- 2.4 之后的重大修订
- 编程模型与其他 EE 规范更趋一致
- 可插拔性与按需加载
  - 将责任交给框架开发人员
- 为 Comet 样式的应用程序提供更好的支持

Hardware and Software
Engineered to Work Together

硬件和软件，集成设计、卓越性能