# 第 1 章　Java EE 简介

本章主要内容

Java EE 是一种思想，是一套规范，也是一种企业级应用的软件架构。本章将概要叙述 Java EE 的思想、规范和架构。具体的内容包括 Java EE 的发展历史，Java EE 平台的体系结构，Java EE 规范第 8 版的新特性，Java EE 的组件/容器的编程思想，Java EE 容器的种类及服务，Java EE 规范定义的组件种类，以及分布式多层应用模型等。本章还简单介绍了本书中所使用的集成环境和配置。

## 1.1　Java EE 的产生及定义

### 1.1.1　Java 的产生

自从 1946 年第一台计算机诞生，软件开发就成了一个重要的主题，因为一项任务的完成需要一定的控制流程和运算步骤来实现。早期的软件开发需要专业背景很强的专家来完成，后来随着计算机应用的普及和编程语言的发展，开始了作坊式的软件开发，这个时期主要实现小规模的软件系统。从程序设计方法学的角度来看，在软件开发的道路上出现了结构化程序设计、面向对象程序设计等。随着软件系统的广泛应用、软件开发规模的增加，导致了软件危机的爆发。人们试图用工程化的方法来解决这个问题，从而出现了软件工程这门学科。随着网络的出现，特别是互联网的出现，人们的生产和生活方式有了很大改变，很多企业借机通过向客户、合作伙伴、员工和供应商提供易于访问的服务的方式来扩大业务范围、降低成本、缩短响应时间等。通常，提供这些服务的应用程序必须整合现有的企业信息系统（enterprise information system，EIS），并增加新业务功能来为更广泛的用户提供服务。这些服务需要满足以下要求：

① 高度可用性，以满足全球商业环境的需要；

② 安全性，以保护用户的隐私和企业数据的完整性；

③ 可靠性和可扩展性，以确保业务交易准确且可扩展，并且处理迅速。

人们把这样的系统称为企业级应用程序，也就是指那些应用规模巨大、集成了很多应用功能、需要处理巨量数据的软件开发项目，一般具有以下特点：

① 是基于网络的应用，而不是基于单机的；

② 巨量的数据集成；

③ 高度的安全性；

④ 具备可扩展性。

目前软件开发中，有基于 C/S（client/server，客户-服务器）体系结构的软件开发和基于 B/S（browse/server，浏览器-服务器）体系结构的软件开发两个方向。基于 C/S 体系结构的系

统由客户端（client）和服务器端（server）两部分构成。用户想要使用这个系统，首先必须先安装它的客户端，比如手机里的 QQ 等软件。客户端负责人机界面的交互及业务控制方面的操作，服务器端主要负责数据的交互和保存。这种系统的优点是系统安全性高，通信效率高，能处理大量数据，交互性强。基于 B/S 体系结构的系统由浏览器（browser）和服务器（server）组成。浏览器只是起到了“浏览”的作用，它仅仅把程序需要传递的页面在浏览器中呈现出来，本身不对数据做任何处理。在这种体系结构中，服务器内部进行了一个分层，应用服务器负责实现业务处理和控制，可以近似认为代替了 C/S 中客户端的部分功能，数据库服务器负责对数据库的管理和对数据的具体交互。这种系统的优点就是客户只需要一个浏览器，不需要安装客户端，而且服务器的分层有效地使程序和数据分离，提高了独立性。

软件开发的过程是一个不断标准化、专业化、抽象化的过程。作为企业级的软件系统开发也需要一个规范化的标准，以便于系统开发。Java EE 就是为了实现这个目标而产生的，也为了适应这一目标在不断地改进。从面向对象到 Java EE，对开发过程中的应用对象作了进一步抽象，对开发过程中各个组件的接口进行了统一，给出了基于 B/S 结构的软件系统开发应遵循的规范，并且定义了用于开发的组件。Java EE 是基于 Java 语言的一种软件设计体系结构。所谓软件体系结构，也就是一个抽象的系统规范，主要包括用其行为来描述的功能构件和构件之间的相互连接、接口和关系，更准确地说是一种标准中间件体系结构。中间件一种独立的系统软件或服务程序，位于 C/S 的操作系统之上，管理计算机资源和网络通信，是连接两个独立应用程序或独立系统的软件。即使相互连接的系统具有不同的接口，它们之间仍能通过中间件进行信息交换。应用程序可以通过中间件在多平台或 OS 环境中运行。

表 1-1 展示了 Java EE 的发展历程及一些改进。

**表 1-1 Java EE 的发展史**

| 时　间 | 版　本 | 新增和改进特性 |
|---|---|---|
| 1999 年 12 月 | J2EE 1.2 | EJB，Servlets，JSP，JMS |
| 2001 年 8 月 | J2EE 1.3 | CMP，JCA |
| 2003 年 11 月 | J2EE 1.4 | JAX/RPC，Management API，部署（deployment） |
| 2006 年 5 月 | Java EE 5 | JSF 1.2，JPA，Annotations，JAXB，JAX-WS，EJB 3.0 |
| 2009 年 12 月 | Java EE 6 | JAX-RS，CDI，依赖注入，Bean Validation，JASPIC，EJB，Servlets 3.0，JSF，Web Profile |
| 2013 年 5 月 | Java EE 7 | 面向 Java 平台的批处理应用，面向 Java EE 的并发工具，用于 JSON 处理的 JavaAPI，WebSocket Java API，<br>EJB 组件，Servlet，JSF 组件，Java 消息服务 |
| 2017 年 8 月 | Java EE 8 | 用于 JSON 绑定的 Java API，Java EE 安全 API，JSON 处理的对象模型，RESTful web 服务，Servlets，JSF 组件，上下文和依赖注入，JavaBean 有效验证 |

多年来，Java EE 一直是企业级应用程序开发的主要平台。为了加速云本地的业务应用开发，多家软件供应商共同协作于 2017 年 9 月把 Java EE 技术迁移到 Eclipse 基金会，并更名为 JakartaEE，致力于指定一组规范，使得全球的 Java 开发者能够在云本地（cloud native）Java 企业应用上开展工作。第一个版本就是 Jakarta EE 8，它基于从 Oracle 转移到 Eclipse 基金会的 Java EE 8 技术。2020 年 9 月份，Eclipse 基金会发布了 Jakarta EE 9.0 版本，是 Eclipse 基金会的首个正式版本，命名空间从 javax.*迁移到 jakarta.*；所有模块都进行了升级，如 Servlet 4.0.2 升级为 Servlet 5。于 2021 年 9 月份又发布了 Jakarta EE 9.1 版本，此版本主要提供对 Java

SE 11 的运行支持，没有新的 API。

尽管 Eclipse 基金会已经接管了 Java EE，并且已经更名为 Jakarta EE，但在一个时期内，Java EE 仍然会在企业级应用系统的开发占据重要的位置。

### 1.1.2　Java EE 的定义

**1．Java EE 是一个应用程序模型**

Java EE 是用于支持客户、员工、供应商、合作伙伴和其他提出要求的人或者对企业有贡献的人完成企业服务活动的应用程序。这类应用程序本身就具有很强的复杂性，需要潜在地访问各种来源的数据，并将应用程序分发到各种客户。

为了更好地控制和管理这些应用程序，Java EE 将企业应用程序划分为多个不同的层，并在每一层上都定义对应的组件来实现。对不同用户的事务管理在中间层实现。中间层是一个由企业信息技术部门紧密控制的环境，通常在专用服务器硬件上运行，并且能访问企业的所有服务。

Java EE 应用程序通常依赖 EIS 层来存储企业的关键业务数据，这些数据和管理这些数据的系统是企业的核心。最初，两层 C/S 应用程序模型为改进可扩展性和功能带来了希望，很遗憾的是，直接向每一位用户传送 EIS 服务的复杂性，以及在每台用户机器上安装和维护事务逻辑带来的管理问题，已被证明了是这种结构的主要局限。

通过将企业服务划分为多层应用程序，可以避免这种两层结构的局限性。多层应用程序提供了现在企业的所有要素都需要的更高的可访问性。这一转变大大驱动了在中间件开发上的投资。

开发多层服务的难度在于同时开发服务事务功能和更复杂的需要去访问数据库和其他系统资源的基础代码。因为每个多层服务器产品有自己的应用模式，很难招聘和培训有经验的开发人员。此外，随着服务量的增加，通常需要更改整个多层基础架构，导致大量的移植成本，从而导致延期。

Java EE 应用程序模型定义了实现服务的多层应用程序的体系结构，可避免如上问题，并提供所需的可扩展性、可访问性和可管理性。

Java EE 应用程序模型把需要完成多层服务的工作划分为两部分：由开发人员实现的事务和表示逻辑，以及 Java EE 平台提供的标准系统服务。开发人员可以依靠该平台为开发中间层服务的硬件系统级别的问题提供解决方案。

Java EE 应用程序模型为多层应用程序提供了一次编写、随时随地运行的可移植性和可扩展性的优点。这个标准模型减少了开发人员培训的成本，同时提供企业在 Java EE 服务器和开发工具方面更广泛的选择。

关于多层应用程序模型将在 1.8 节“企业级应用程序体系结构”中进一步介绍。

**2．Java EE 是企业分布式应用程序开发的一组规范**

Java EE 给出了企业级应用程序开发的规范，主要包含以下内容。

① Java EE 平台的 Java 技术标准。Java EE 平台的主要组成部分包含所有 Java EE 产品都需要支持的 Java 技术标准。由于 Java EE 平台聚焦的是企业解决方案的端到端开发，它不仅仅要求支持每个 Java API，而且要求每个 API 必须完全集成到平台中，确保平台为 Java EE 应

用的部署提供一致的端到端环境。

② Java EE 平台的 CORBA 技术标准。对象管理组（object management group，OMG）与 Sun 共同制定了 RMI-IIOP 规范。该标准定义了 Java 远程方法调用工具如何使用 CORBA IIOP 协议。EJB 规范使用 RMI-IIOP 的应用程序映射作为调用 EJB 的标准。Java EE 平台强烈支持使用 RMI-IIOP。

③ Java EE 平台的 IETF 标准。互联网的出现对企业应用程序的开发产生了重大影响。这场革命是基于互联网工程任务组（Internet Engineering Task Force，IETF）制定的标准，包括 HTML、HTTP，以及 XML 和通信结构化数据的 Internet 标准。

Java 程序设计语言跟随着 IETF 标准一起成长，并成为人们编写应用程序的首选方式。Java EE 应用程序模型和 Java EE 平台延续了这一趋势。Java EE 平台支持 HTML 和 HTTP 客户端，以及 XML 客户端。此外，Java EE 部署描述符利用 XML 以独立于平台的方式提供应用程序信息。

④ Java EE 部署规范。这是一个定义通用部署方式的标准。Java EE 应用程序被打包成一个或多个标准单元，以便部署到任何符合 Java EE 兼容性平台的系统上。每个单元包含一个功能组件或多个组件（企业 Bean、JSP 页面、Servlet、Applet 等）；一个标准部署描述符，用以描述其内容和由应用程序开发人员和汇编器描述的 Java EE 声明。部署通常涉及使用平台的部署工具来指定特定位置信息，例如可以访问它的本地用户列表和本地数据库的名称。一旦在本地平台上部署完毕，应用程序就准备运行了。

开发人员可以从网址 https://jcp.org/en,jsr:detai?d=366 下载最新的 Java EE 8 规范，它包含了 32 个具体的标准。

需要强调的是，Java EE 规范只是一个标准，它不定义组件和容器的具体实现。容器由第三方厂商来实现（如 Oracle、IBM 等），这通常被称为应用服务器。而组件由开发人员根据具体的业务需求来实现，各种不同类型的组件部署在容器里，最终构成 Java EE 企业应用系统。

尽管不同的厂商由不同的容器产品实现，但它们都遵循 Java EE 规范，因此遵循 Java EE 标准的组件，可以自由部署在这些由不同厂商生产、但又相互兼容的 Java EE 容器环境中，企业级系统的开发由此变得简单高效。

**3．Java EE 兼容性测试套件**

Java EE 平台供应商需要验证其实现是否符合 Java EE 平台规范。为了达到这个目的，Sun Microsystems 公司授权平台供应商使用 Java EE 兼容性测试套件（compatibility test suite，CTS）。

开发商通过其 GUI 框架将此测试套件部署、配置和运行他们的平台实现上。该套件包含多个测试，以确保实现了 Java EE API。测试将验证 Java EE 组件技术是可用的，并且能够正确地协同工作。该套件还包括一套功能齐全的 Java EE 应用程序，以验证所有平台都能够协调地部署和运行它们。

**4．Java EE 参考实现**

Java EE 参考实现完成了多个角色。它的主要角色就是 Java EE 平台可操作的定义。在这个角色中，它是供应商将其用作 Java EE 平台的必需标准，以确定其实现必须在一组特定的应用环境下进行。它也被开发人员用来验证应用程序的可移植性。最重要的是，它用作运行 Java EE 兼容性测试套件的标准平台。

参考实现的第二个角色是可用于推广 Java EE 平台（企业版）的平台。虽然不是商业产品，其许可条款将禁止其商业用途，它将以二进制形式免费提供，用于演示、原型设计和测

试学术研究。参考实现也以源代码形式提供。

## 1.2　Java EE 平台的体系结构

图 1-1 展示了 Java EE 平台体系结构中各元素之间满足的关系。此图展示的是元素间的逻辑关系，它并不代表这些元素在物理上划分为不同的机器、进程、地址空间或虚拟机。

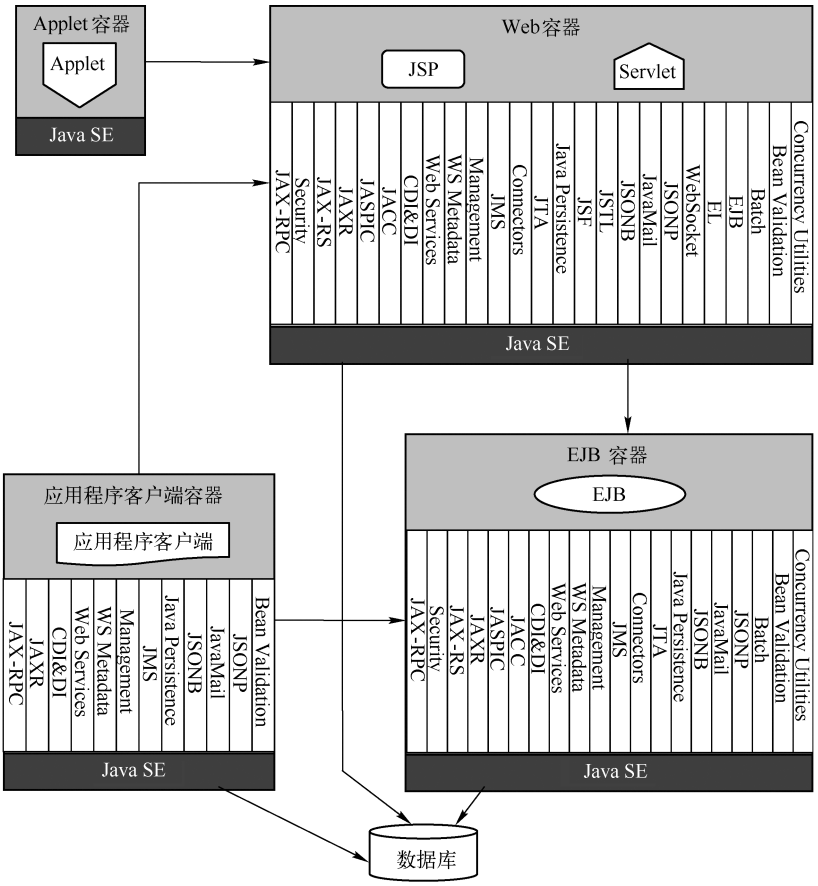


图 1-1　Java EE 平台体系结构示意图

容器由独立矩形表示，它是 Java EE 运行环境，为矩形框上半部分中描述的应用组件提供必需的服务，提供的服务在矩形的下半部分的格子中进行了标注。例如，应用程序客户端容器向应用程序客户端提供 Java 消息服务（Java meseage service，JMS）的 API 接口，其他服务也是如此。箭头描述了 Java EE 平台中的一部分对其他部分必需的访问。应用程序客户端容器提供了直接访问 Java EE 必需的数据库的 Java API 接口，JDBC™ API。类似地，对数据库的访问还应该由 Web 容器提供给 JSP 页面、JSF 应用和 Servlet，以及由 EJB 容器提供给 EJB。

Java™ 平台标准版（Java SE）的 API 给每种类型应用组件提供了 Java SE 运行环境。

## 1.3　Java EE 8 的新特性

Java EE 8 平台的主要目标是使云和微服务环境中的企业 Java 基础设施现代化，强调

HTML5 和 HTTP/2 支持，通过新的上下文和依赖注入功能增强开发的易用性，并进一步增强平台的安全性和可靠性。相较以往的版本，Java EE 8 平台增加了以下新功能。

### 1.3.1 新增加的技术

**1．用于 JSON 绑定的 Java API：JSON-Binding 1.0 API (JSON-B)**

（1）JSON-B 为 JSON 序列化和反序列化提供了本地 Java EE 解决方案

以前，如果想要对 JSON 进行序列化和反序列化，就必须依赖 Jackson、GSON 等第三方 API，现在可以使用 JSON-B。从 Java 对象生成 JSON 文档现在非常简单了，只需调用 toJson() 方法并将它传递给想要序列化的实例即可。例如：

```
String bookJson = JsonbBuilder.create().toJson(book);
```

将 JSON 文档反序列化为 Java 对象也非常简单，只需将 JSON 文档和目标类传递给 fromJson()方法，然后返回 Java 对象。例如：

```
Book book = JsonbBuilder.create().fromJson(bookJson, Book.class);
```

（2）定制 JSON-B

通过注释字段、JavaBeans 方法和类，可以自定义 Jsonb 方法的默认行为。

例如，可以使用@JsonbNillable 自定义 null 处理；使用@JsonbPropertyOrder 注释自定义属性顺序，这是需要在类级别进行说明的。可以使用@JsonbNumberFormat()注释指定数字格式，并使用@JsonbProperty()注释更改字段的名称。

```
@JsonbNillable
@JsonbPropertyOrder(PropertyOrderStrategy.REVERSE)
public class Booklet {
    private String title;
    @JsonbProperty("cost")
    @JsonbNumberFormat("#0.00")
    private Float price;
    private Author author;
    @JsonbTransient
    public String getTitle() {
        return title;
    }
    @JsonbTransient
    public void setTitle(String title) {
        this.title = title;
    }
    // price and author getters/setter removed for brevity
 }
```

或者选择使用运行时配置生成器 JsonbConfig 处理自定义。

```
JsonbConfig jsonbConfig = new JsonbConfig()
    .withPropertyNamingStrategy(PropertyNamingStrategy.LOWER_CASE_WITH_DASHES)
```

```
    .withNullValues(true).withFormatting(true);
Jsonb jsonb = JsonbBuilder.create(jsonbConfig);
```

无论哪种方式，JSON 绑定 API 都为 Java 对象的序列化和反序列化提供了解决方案。

（3）开源绑定（open source binding）

下面的程序说明了如何配置实现开源绑定。

```
JsonBuilder builder = JsonBuilder.newBuilder("aProvider");
```

**2．Java EE 安全 API：Java EE Security 1.0 API**

Java EE 8 中增加的最有意义的新特征就是新的安全 API。引入新的 Java EE 安全 API 是为了修正在 servlet 容器之间实现安全问题的不一致性。这个问题在 JavaWeb 配置文件中特别明显，主要是因为 Java EE 只要求完整的 Java EE 配置文件必须实现标准 API。新规范引入了 CDI 等新功能。Java EE 安全 API 规范解决了如下三个关键问题。下面对它们做简单介绍。

（1）注释驱动的身份验证

Java EE 已经为验证 Web 应用程序用户制定了两种机制：Java Servlet 规范 3.1（JSR-340）为应用程序配置制定了声明性机制，而容器的 Java 身份认证服务提供者接口（Java authentication service provider interface for containers，JASPIC）定义了一个称为 ServerAuthModule 的 SPI，它支持处理任何证件类型的身份验证模块的开发。

这两种机制都是有意义和有效的，但从 Web 应用程序开发人员的角度来看，每种机制都是有限制的。servlet 容器机制仅限于支持一小部分可信类型。尽管 JASPIC 非常强大和灵活，但它的使用也相当复杂。

Java EE 安全 API 试图通过一个新接口 HttpAuthenticationMechanism 解决这些问题。本质上，它是 JASPIC 的 ServerAuthModule 接口的一个简化的 servlet 容器变种，它提升了现有的机制，同时降低了它们的限制。

HttpAuthenticationMechanism 类型的一个实例是一个 CDI Bean，它可用于容器注入，并且仅适用于 servlet 容器，明确排除了 EJB 和 JMS 等其他容器。

HttpAuthenticationMechanism 接口定义了三种方法：validateRequest()、secureResponse() 和 cleanSubject()。这些方法与 JASPIC ServerAuth 接口的声明方法非常相似，唯一需要重写的方法是 validateRequest()；所有其他方法都有默认实现。

HttpAuthenticationMechanism 接口，带有三个内置的启用 CDI 的实现，每个实现代表 Web 安全性可配置的三种方式之一。3 个新注释如下。

```
@BasicAuthenticationMechanismDefinition（基本身份认证机制定义）;
@FormAuthenticationMechanismDefinition（表单身份认证机制定义）;
@CustomFormAuthenticationMechanismDefinition（自定义表单身份认证机制定义）。
```

例如，要启用基本身份认证，需要做的是将 BasicAuthenticationMechanismDefinition 注释添加到 Servlet 中，如下面的程序所示。

```
@BasicAuthenticationMechanismDefinition(realmName="${'user-realm'}")
@WebServlet("/user")
@DeclareRoles({ "admin", "user", "demo" })
@ServletSecurity(@HttpConstraint(rolesAllowed = "user"))
```

```
public class UserServlet extends HttpServlet {
//其他代码
}
```

（2）身份存储抽象

身份存储是一个存储用户身份数据的数据库，例如用户名、组成员身份和用于验证凭证的信息。新的 Java EE 安全 API 提供了一个名为 IdentityStore 的身份存储抽象，用于与身份存储交互以验证用户身份和检索组成员身份，类似于 JAAS LoginModule 接口。设计 IdentityStore 的目的是为了接口 HttpAuthenticationMechanism 的实现，并且大多数应用场景也推荐同时使用 IdentityStore 和 HttpAuthenticationMechanism，使得应用程序能够以可移植的标准方式控制身份存储验证。但 IdentityStore 也可以独立使用，并由应用程序开发人员希望的任何其他身份验证机制使用。可以通过实现 IdentityStore 接口来实现自己的身份存储，也可以将内置的 IdentityStore 实现之一用于 LDAP 和关系数据库。通过将配置详细信息传递给适当的注释@LdapIdentityStoreDefinition 或@DataBaseIdentityStoreDefinition 实现初始化。

下面的例子说明了内置身份存储的使用方法。最简单的身份存储是数据库存储，通过@DataBaseIdentityStoreDefinition 注释进行配置，如下面的程序所示。

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup = "${'java:global/permissions_db'}",
  callerQuery = "#{'select password from caller where name = ?'}",
  groupsQuery = "select group_name from caller_groups where caller_name = ?",
  hashAlgorithm = PasswordHash.class,
  priority = 10
)
@ApplicationScoped
@Named
public class ApplicationConfig {
//其他代码
}
```

请注意优先级设置为 10。这用于运行时找到多个身份存储的情况，并确定相对于其他存储的迭代顺序时使用。数字越小，优先级越高。

（3）安全上下文（security context）

安全上下文的目标是跨 servlet 和 EJB 容器提供对安全上下文的一致访问。目前，这些容器实现的安全上下文对象是不一致。例如，servlet 容器提供了一个 HttpServletRequest 实例，在该实例上调用 getUserPrincipal()方法以获取用户主体，而 EJB 容器提供了不同名称的 EJBContext 实例，在该实例上调用了相同的命名方法。同样，为了测试用户是否属于某个角色，在 HttpServletRequest 实例上调用方法 isUserRole()，在 EJBContext 实例上调用 isCallerRole()。

SecurityContext 为跨 Servlet 和 EJB 容器获取此类信息提供了一致性。它有五个方法，并且都没有默认实现。

第一种方法是 Principal getCallerPrincipal()，返回当前已验证用户名表示的特定平台主体，如果当前调用方未被授权，则返回 null。

第二种方法是<T extends principal> Set <T> getPrincipalsByType(Class<T>pType)，从经过身份验证的调用方的主题返回给定类型的所有主体。如果找不到 pType 类型或当前用户未通过身份验证，则返回空集。

第三种方法是 Boolean IsCallerRole(String role)，确定调用方是否包含在指定角色中。如果用户未经授权，则返回 false。

第四种方法是 bollean hasAccessTowereSource(String resource, string, …methods)，确定调用方是否可以通过提供的方法访问给定的 Web 资源。

第五种方法是 AuthenticationStatus authenticate(HttpServletRequest req, HttpServletResponse res, AuthenticationParameters param)，通知容器应启动或继续与调用方进行基于 HTTP 的身份验证对话。此方法仅在 servlet 容器中使用，因为它依赖于 HttpServletRequest 和 HttpServlet-Response 实例。

SecurityContext 是 CDI Bean，因此可以注入 servlet 和 EJB 容器中的任何类。

```
@Inject
private SecurityContext securityContext;
```

有了 SecurityContext 实例，就可以调用任何方法来访问上下文感知的安全信息。

```
boolean hasAccess = securityContext.hasAccessToWebResource("/secretServlet", "GET");
```

## 1.3.2　改进的技术

### 1．Servlet 的新特性

Java EE 8 中给出了 Servlet 4.0 规范，做了一些改进，具体如下所述。

（1）服务器推送

Servlet 4.0 规范中定义了服务器推送功能，以便与 HTTP/2 协议保持一致。

服务器推送是 HTTP/2 协议中的新特性之一，旨在通过将服务器端的资源推送到浏览器的缓存中来预测客户端的资源需求，以便当客户端发送网页请求并接收来自服务器的响应时，它需要的资源已经在缓存中。这是一项提高网页加载速度的性能增强的功能。

在 Servlet 4.0 中，服务器推送功能是通过 PushBuilder 实例实施的，此实例是从 HttpServletRequest 实例中获取的。

由下面这个代码片段可以看到 header.png 的路径通过 path()方法设置在 PushBuilder 实例上，并通过调用 push()方法被推送到客户端。当方法返回时，路径和条件报头将被清除，为构建器重用做好准备。然后推送 menu.css 文件，接着是推送 ajax.js 这个 JavaScript 文件。

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    PushBuilder pushBuilder = request.newPushBuilder();
    pushBuilder.path("images/header.png").push();
    pushBuilder.path("css/menu.css").push();
    pushBuilder.path("js/ajax.js").push();
    // Do some processing and return JSP that requires these resources
}
```

当 Servlet 的 doGet()方法执行完毕后，资源将会到达浏览器。从 JSP 生成的 HTML 需要这些资源，但不需要从服务器请求它们，因为它们已经在浏览器的缓存中。

（2）servlet 映射的运行时发现

Servlet 4.0 提供了一个用于 URL 映射的运行时发现的新 API。HttpServletMapping 接口的目标是更容易确定导致 servlet 被激活的映射。在 API 内部，从 HttpServletRequest 实例获得 Servlet 映射有以下 4 种方法。

① getMappingMatch()：返回 match 的类型。

② getPattern()：返回激活 servlet 请求的 URL 模式。

③ getMatchValue()：返回匹配的字符串。

④ getServletName()：返回请求激活的 servlet 类的完全限定名。

下面是一个示例。

```
HttpServletMapping mapping = request.getHttpServletMapping();
String mapping = mapping.getMappingMatch().name();
String value = mapping.getMatchValue();
String pattern = mapping.getPattern();
String servletName = mapping.getServletName();
```

除了上面的更新，Servlet 4.0 还对内务管理做了少量更新，并提供了对 HTTP trailer 的支持。新的 GenericFilter 和 HttpFilter 类简化了编写过滤器的工作，并实现了对 Java SE 8 的全面升级。

**2．Bean 验证**

Bean 验证规范定义了一个元数据模型和 API，用于验证 JavaBeans 组件中的数据。可以在一个位置定义验证约束，并在不同的层之间共享这些约束，而不是在多个层（如浏览器和服务器端）上分发数据验证。在 Java EE 8 平台中，新的 Bean 验证功能包括以下内容。

（1）支持 Java SE 8 中的新特性，如日期时间 API 等

更新后的 Bean 验证 API 提升了 Java SE 8 的日期和时间类型，并提供了对 Java.util.Optional 的支持，如下所示。

```
Optional<@Size(min = 10) String> title;
private @PastOrPresent Year released;
private @FutureOrPresent LocalDate nextVersionRelease;
private @Past LocalDate publishedDate;
```

（2）添加新的内置 Bean 验证约束

BeanValidation 2.0 通过一系列新特性得到了增强，其中许多特性是 Java 开发人员社区要求的。Bean 验证是一个多方关注的问题，因此 2.0 规范通过对字段、返回值和方法参数中的值使用约束来确保从客户端到数据库的数据完整性。

增强的功能包括验证电子邮件地址、确保数字为正数或负数、测试日期是否为过去或现在，以及测试字段是否为空或 null。它们是：@Email、@Positive、@PositiveOrZero、@Negative、@NegativeOrZero、@PastOrPresent、@FutureOrPresent、@NotEmpty 和@NotBlank。约束现在也可以在更广泛的地方发挥作用，例如，它们可以继续参数化类型的变量。增加支持按类型参数验证容器元素，如下所示。

```
private List<@Size(min = 30) String> chapterTitles;
```

（3）容器的级联验证

用@Valid 注释容器的任何类型参数将导致在验证父对象时验证每个元素。在下面的程序中，每个 String 和 Book 元素都将进行验证。

```
Map<@Valid String, @Valid Book> otherBooksByAuthor;
```

Bean 验证还可以通过插入值提取器来增加对自定义容器类型的支持。内置约束被标记为可重复的，参数名使用反射检索，默认方法是 ConstraintValidator#initialize()。

**3．上下文和依赖注入的新特性**

上下文和依赖项注入 API（contexts and dependency injection，CDI）自第 6 版在 Java EE 中使用后就成为一种重要技术。从那时起，它因为使用方便而成为 Java EE 的主要特征。CDI 2.0 还对观察器（observers）、事件行为（events behave）和交互方式（interact）进行了重要修正。

（1）CDI 2.0 中的观察器和事件行为

在 CDI 1.1 中，当一个事件被触发时，观察器被同步调用，没有机制来定义它们的执行顺序。此行为的问题在于，如果观察器抛出异常，则不会调用所有后续观察器，并且观察器链停止。在 CDI 2.0 中，通过引入@Priority 注释，在一定程度上缓解了这种情况，该注释指定了调用观察器的顺序，首先调用优先级较小的观察器。

下面的程序显示了一个事件触发和两个优先级不同的观察器的情况，在观察器 AuditEvent-ReceiveR2（优先级 100）之前调用观察器 AuditEventReceiveR1（优先级 10）。

```
@Inject
private Event<AuditEvent> event;
public void send(AuditEvent auditEvent) {    event.fire(auditEvent); }
// AuditEventReciever1.class
public void receive(@Observes @Priority(10) AuditEvent auditEvent) {
      // react to event
}
// AuditEventReciever2.class
public void receive(@Observes @Priority(100) AuditEvent auditEvent) {
// react to event
}
```

需要注意的是，具有相同优先级的观察器以不可预测的顺序被调用，默认顺序是 javax.interceptor.interceptor.priority.APPLICATION+500。

（2）异步事件

观察器的另一个附加功能是异步触发事件的能力。新添加的触发方法（fireAsync()）和相应的观察器注释（@ObservesAsync）用以支持此功能。下面的程序说明了被异步触发的 AuditEvent，以及接收事件通知的观察器方法。

```
@Inject
private Event<AuditEvent> event;
public CompletionStage<AuditEvent> sendAsync(AuditEvent auditEvent) {
```

```
    return event.fireAsync(auditEvent);
}
// AuditEventReciever1.class
public void receiveAsync(@ObservesAsync AuditEvent auditEvent) {}
// AuditEventReciever2.class
public void receiveAsync(@ObservesAsync AuditEvent auditEvent) {}
```

如果任何观察器引发异常，则 CompletionStage 将在 CompletionException 中完成。此实例包含对观察器调用期间抛出的所有抑制异常的引用。下面的程序展示了如何管理这种情况。

```
public CompletionStage<AuditEvent> sendAsync(AuditEvent auditEvent) {
    System.out.println("Sending async");
    CompletionStage<AuditEvent> stage = event.fireAsync(auditEvent)
            .handle((event, ex) -> {
                if (event != null) {
                    return event;
                } else {
                    for (Throwable t : ex.getSuppressed()) {}
                    return auditEvent;
                }
            });
    return stage;
}
```

如前所述，CDI 2.0 还获得了对 Java SE 8 特性的提升，如流、λ表达式和可重复限定符。还有其他几个方面做了改进：

① 一个新的配置器接口；

② 配置或否决观察器方法的能力；

③ 内置注释文本；

④ 在生产者上应用拦截器的能力。

**4．RESTful Web 服务的新特性**

JAX-RS 2.1 版本的 API 关注两个主要特性：一个新的反应式客户端 API 和对服务器发送事件的支持。

（1）反应式客户端 API

RESTful Web 服务自 1.1 版以来就包含了一个客户端 API，提供了对 Web 资源访问的高级别方法。JAX-RS 2.1 中，此 API 支持对反应式编程。最明显的区别是用于构造客户端实例的 Invocation.Builder。从下面的程序可以看出，新方法 rx()以 Response 为参数类型，返回类型为 CompletionStage。

```
CompletionStage<Response> cs1 = ClientBuilder.newClient()
        .target("http://localhost:8080/jax-rs-2-1/books")
        .request()
        .rx()
        .get();
```

CompletionStage 接口是在 Java 8 中引入的，由此带来了很多有意义的结果。例如，在如下的代码段中，对不同端点进行了两次调用，然后将结果组合到一起。

```
CompletionStage<Response> cs1=ClientBuilder.newClient().target(".../books/history").request().rx().get();
CompletionStage<Response> cs2=ClientBuilder.newClient().target(".../books/geology").request().rx().get();
cs1.thenCombine(cs2,(r1,r2)->r1.readEntity(String.class)+r2.readEntity(String.class))
                                .thenAccept(System.out::println);
```

（2）对服务器发送事件的支持

服务器发送事件 API（server-sent events，SSE）由 W3C 在 HTML 5 中引入，并由 WHATWG 社区维护，允许客户端订阅服务器生成的事件。在 SSE 体系结构中，创建了从服务器到客户端的单向通道，通过该通道，服务器可以发送多个事件。该连接持续时间长，并保持打开状态，直到任一侧将其关闭。

JAX-RS API 包括 SSE 的客户端和服务器 API。客户端的入口点是 SseEventSource 接口，该接口使用已配置的 WebTarget 进行修改，如下列程序所示。

```
WebTarget target = ClientBuilder.newClient().target("http://localhost:8080/jax-rs-2-1/sse/");
try (SseEventSource source = SseEventSource.target(target).build()) {
        source.register(System.out::println);
        source.open();
}
```

在此代码段中，客户机注册一个使用者。使用者输出到控制台，然后打开连接。还能支持 onComplete 和 onError 生命周期事件的处理程序。

在另一端，SSE 服务器 API 接收来自客户端的连接，并向所有连接的客户端发送事件。

```
@POST
@Path("progress/{report_id}")
@Produces(MediaType.SERVER_SENT_EVENTS)
public void eventStream(@PathParam("report_id")String id,
                        @Context SseEventSink es,
                        @Context Sse sse) {
    executorService.execute(() -> {
    try {
        eventSink.send(
            sse.newEventBuilder().name("report-progress")
                .data(String.class, "Commencing process for report " + id).build());
            es.send(sse.newEvent("Progress", "25%"));
            Thread.sleep(500);
            es.send(sse.newEvent("Progress", "50%"));
            Thread.sleep(500);
            es.send(sse.newEvent("Progress", "75%"));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
        });
    }
```

在上述程序中，SseEventSink 和 Sse 资源被注入资源方法中。Sse 实例获取一个新的出站事件生成器，并向客户端发送进度事件。需要注意的是，媒体类型是一种新的文本/事件流类型，专门用于事件流。

（3）广播服务器发送事件

事件可以同时广播到多个客户端。这是通过在 SseBroadcaster 上注册多个 SseEventSink 实例来实现的。

```
@Path("/")
@Singleton
public class SseResource {
    @Context
    private Sse sse;
    private SseBroadcaster broadcaster;
    @PostConstruct
    public void initialise() {
        this.broadcaster = sse.newBroadcaster();
    }
    @GET
    @Path("subscribe")
    @Produces(MediaType.SERVER_SENT_EVENTS)
    public void subscribe(@Context SseEventSink eventSink) {
        eventSink.send(sse.newEvent("You are subscribed"));
        broadcaster.register(eventSink);
    }
    @POST
    @Path("broadcast")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void broadcast(@FormParam("message") String message) {
        broadcaster.broadcast(sse.newEvent(message));
    }
}
```

上述程序中的示例接收 URI/subscribe 上的订阅请求。订阅请求引发为该订阅服务器创建 SseEventSink 实例，然后向 SseBroadcaster 注册该资源。向所有订阅者广播的消息是从提交到 URL/broadcast 的网页表单中检索到，然后由 broadcast()方法处理。

**5．JSON 处理的新特征**

JSON（JavaScript 对象表示法）是许多 Web 服务使用的轻量级数据交换格式。用于 JSON 处理的 Java API（JSON-P）提供了一种方便的方式来处理（解析、生成、转换和查询）JSON 文本。在完整的 Java EE 产品中，所有 Java EE 应用程序客户端容器、Web 容器和 EJB 容器都

需要支持 JSON-PAPI。在 Java EE 8 平台中，发布了 JSON-P 的一个版本，使其符合最新的 IEFT 标准，在如下 4 个方面做了改进。

（1）JSON 指针

JSON 指针是 JSON 处理 1.1 API 中的一个新功能，并可以使用最新的 IEFT 标准 JSON 指针进行更新。

JSON 指针定义了一个字符串表达式，该表达式引用 JSON 文档的层次结构内的元素。使用 JSON 指针表达式，可以通过检索、添加、删除和替换表达式引用的元素或值来访问和操作 JSON 文档。API 入口是接口 javax.json.JsonPointer。通过在 javax.json.Json 类上调用静态工厂方法 createPointer(String expression)，并向其传递指针表达式来创建实例。

例如，有如下的一个 JSON 文档：

```
{ "topics":
    "Cognitive",
    "Cloud",
    "Data",
    "IoT",
    "Java" }
    JsonObject jsonObject = ... create JsonObject from JSON document ...;
```

如果想检索 title 元素的值，则用下面的代码段创建一个 JsonPointer 对象，并引用 title 元素，然后，调用 getValue()方法，并将 JsonObject 的实例传递给该方法进行查询。

```
JsonValue jsonValue = Json.createPointer("/topics/1").getValue(jsonObject);
```

要向 JSON 文档添加（或插入）值，遵循与检索相同的方法，即使用 JSON 指针表达式来标识文档内的插入点。以下代码段将新的“Big Data”JSON 对象添加到 jsonObject 中。

```
Json.createPointer("/topics/0").add(jsonObject , Json.createValue("Big Data"));
```

返回的 JsonObject 是修改后的对象。类似的，还有删除和替换操作。

（2）JSON 补丁

JSON 补丁表示一个操作的序列，这些操作应用于目标 JSON 文档，包括添加、复制、移动、删除、替换和测试等操作。

JsonPatchBuilder 接口是该 API 的关键，它由 Json 类上的静态方法 createPatchBuilder()创建。JSON 指针表达式被传递给其中一个操作方法，并应用于 JSON 文档。下列的程序展示了将 topics 数组的第一个元素替换为值“Spring 5”：

```
JsonPatchBuilder builder = Json.createPatchBuilder();
JsonPatch jsonPatch = builder.replace("/topics/0", " Spring 5").build();
JsonObject newJsonObject = jsonPatch.apply(jsonObject);
```

可以顺序执行多个操作，顺序应用于上一个补丁程序的结果。

（3）JSON 合并补丁

JSON 合并补丁是一个 JSON 文档，它描述了对目标 JSON 文档做出更改的集合。表 1-2 显示了三种可能的操作。

表 1-2 合并补丁的操作

| Operation | Target | Patch | Result |
|---|---|---|---|
| Replace | {“color”:”blue”} | {“color”:”red”} | {“color”:”red”} |
| Add | {“color”:”blue”} | {“color”:”red”} | {“color”: null} |
| Remove | {“color”:”red”} | {“color”:”blue”,”color”:”red”} | {} |

Json 类上的静态方法 createMergePatch()用于产生类型为 JsonMergePatch 的一个实例，可以将补丁传递给该实例。然后，将目标 JSON 传递给生成的 JsonMergePatch 实例的 apply()方法即可。下面的程序展示了如何执行表 1-2 中的 replace 操作。

```
Json.createMergePatch(Json.createValue("{\"colour\":\"blue\"}"))
    .apply(Json.createValue("{\"colour\":\"red\"}"));
```

（4）JsonCollectors

随着 Java 8 在 javax.JSON.streams 包中引入 JsonCollectors 类，查询 JSON 变得更加简单。下面的程序展示了按字母 C 过滤 topics 数组，并将结果放到一个 JsonArray 中的过程。

```
JsonArray topics = jsonObject.getJsonArray("topics").stream()
                              .filter(jv -> ((JsonString) jv).getString().startsWith("C"))
                              .collect(JsonCollectors.toJsonArray());
```

**6．JavaServerFaces 组件的新特性**

在 Java EE 8 中，JavaServer Faces 升级到 2.3 版本（JSF 2.3），新特性包括如下几个方面。

（1）改进的 CDI 对齐（alignment）

也许 JSF 2.3 最受欢迎和最有用的补充之一是 CDI 对齐的改进。JSF 2.3 做了很多改进，使得许多 JSF 组件现在可以轻松地注入 Java 类和 EL 表达式中。多年来，JSF 一直使用静态入口方法，并形成了一个诸如 FacesContext、RequestMap 或 FlowMap 的组件链，这些组件现在可以很容易地注入类中，而不是实例化。曾经的命令：

```
FacesContext facesContext = FacesContext.getCurrentInstance();
```

在 JSF 2.3 中，可以简化为如下操作：

```
@Inject
FacesContext facesContext;
```

如果在 JSF 视图中的 EL 表达式的上下文中使用，可以执行以下操作：

```
#{facesContext}
```

有许多元素确实需要 CDI 限定符。例如，流程图可以按如下方式注入：

```
@Inject
@FlowMap
private Map<Object, Object> flowMap;
```

JSF 的一个有用特性是易于开发转换器（convertors）、验证器（validators）和行为器（behaviors）。在 JSF 2.3 中，现在可以将 CDI 组件注入这些目标中，使它们更易于创建。JSF

托管 Bean 的作用域在过去几年中有点混乱，因为它可以同时把托管 Bean 注释和 CDI 用于作用域。随着 JSF 2.3 的发布，托管 Bean 注释被弃用，CDI 成为首选，这将有助于防止开发人员在混合使用托管 Bean 和 CDI 作用域注释时带来错误，因为它们不能很好地组合使用。因此，@ManagedProperty 注释已被弃用。此版本引入了更新的@ManagedProperty 注释，允许保留相同的功能。

```
@Inject
@ManagedProperty ("#{bean.property}")
private String stringProperty;
```

（2）网络和 WebSocket 的更新

WebSocket 是通过 TCP 提供全双工双向通信的协议。它在不断增长的异步环境中已成为标准通信协议。虽然 JSF 在 WebSocket 通信方面已经很好地工作了一段时间，JSF 2.3 完全支持它，通过新的<f:WebSocket>标记使 WebSocket 更便捷。可以将新标记放置到视图中，以便将服务器端通信推送到包含相同通道名称的套接字的所有实例。当收到通信时，可以调用 onmessage JavaScript 事件处理程序，提供客户端功能。使用新标记的示例如下所示，仅在收到消息时显示警报：

```
<h:body>
<f:websocket channel="jaxArticle"onmessage="function(message){alert(mesage)}"/>
</h:body>
```

WebSocket 通信的服务器端能够推出消息。JSF 2.3 添加了 javax.faces.push.PushContext，这是一个可注入的上下文，允许服务器推送到指定的通道。

JSF 2.3 中还有许多其他网络和 AJAX 增强功能，包括如下几个方面。

① 使用新的<h:commandScript/>组件从视图中执行任意服务器端方法。

② 通过 PartialViewContext 从服务器端方法在视图中执行 JavaScript。

③ 通过新引入的“名称空间”模式使用 AJAX 更新多个表单。

（3）验证和转换增强功能的更新

在 JSF 2.3 中，有许多有用的有效性验证，并增强了转换能力，包括更好的 Java SE 8 对齐和验证全部 Bean 有效性的能力。通过使用新的<f:validateWholeBean>标记，类级的 Bean 有效性验证成为可能。通过在视图中包含此标记，并用 Bean 有效性验证组标记至少一个输入组件，可以验证全部 Bean 的有效性，从而启用新的有效性验证支持，例如跨字段有效性验证。

自从 Java SE 8 发布以来，Java 世界的很多地方都在使用新的日期时间 API。Java EE 世界必须使用变通方法才能使这个新 API 正常工作，JSF 2.3 版本向前推进，增强了<f:convertDateTime>转换标记以支持新的日期时间类型。例如，要利用 java.time.LocalDate 类，可以在视图中使用以下内容：

```
<h:outputText value="#{jobController.selected.workDate}">
   <f:convertDateTime type="localDate" pattern="MM/dd/yyyy"/>
</h:outputText>
```

<f:convertDateTime>已经得到了加强，能够支持如下的日期和时间类型：LocalDate、LocalTime、LocalDateTime、OffsetTime、OffsetDateTime 和 ZonedDateTime。

（4）组件和 API 的增强

自 JSF 诞生以来，各种组件和 API 之间存在许多细微差别，这有时会使开发变得烦琐。为了缓解其中一些难点，对一些底层 API 和组件进行了增强。例如，UIData 和 UIRepeat 已得到增强，以支持 Map 和 Iterable 接口以及自定义类型。这些增强功能能够在 DataTables 和 ui:repeat 标记中支持 java.util.Iterable 类型、映射和自定义数据类型。例如，DataTable 中的映射支持类似于以下代码：

```
<h:dataTable var="mapEntry" value="#{jobController.jobMap}">
  <h:column value="#{mapEntry.key}"/>
  <h:column value="#{mapEntry.value}"/>
</h:dataTable>
```

还有许多其他有用的增强功能，包括通过新标记利用 EL 中的常量的能力等。

## 1.4 Java EE 的编程思想

为满足开发多层体系结构的企业级应用程序的需求，Java EE 编程的基本思想就是组件/容器。Java EE 应用程序的基本软件单元是组件，所有的组件都运行在特定的运行环境中，这个运行环境被称为容器。从逻辑上说就是把完成具体功能的工具以组件的形式“装入”一个容器之中，组件对所有数据的接收和发送必须通过容器才能完成，也就是说服务器发出的请求是由容器分发到响应组件进行处理，处理的结果又交给容器，由容器决定最后的输出方式。这样做可以用统一的标准处理数据，并且让容器与系统其他部分保持很高的独立性。这个思想是面向对象中类的封装性的一种延续，只不过类封装的是一个特定的数据模式，容器封装的是更大的一组特定的功能。

Java EE 的容器需要提供必需的底层基础功能来完成组件与外界的数据互交，这些底层基础功能被称为服务，组件又是通过调用容器提供的标准服务来与外界交互的。为满足企业级应用灵活部署，组件与容器之间必须既松散耦合，又能够高效交互。为实现这一点，组件与容器都要遵循一个标准规范。Java EE 提供了一个统一的标准规范，实现了组件与容器之间既保持相对独立，又有比较强大的数据交互能力。

Java EE 容器的具体实现是由专门的厂商来完成的，Java EE 规范只是给出了容器必须实现的基本接口和功能，具体如何实现完全由容器厂商自己决定。常见的 Java EE 服务器中都包含 Web 容器或 EJB 容器的实现。它们所包含的组件会在容器的 Java 虚拟机中进行初始化。组件通过容器提供的标准服务来与外界进行互交，这些服务主要包括命名服务、数据库连接服务、持久化、消息服务、事务支持和安全性服务等。大部分底层的基础功能都由容器提供，可以让开发者专注于对业务逻辑的设计。因此在分布式组件的开发过程中，完全可以不考虑复杂多变的分布式计算环境，而专注于业务逻辑的实现，这样可大大提高组件开发的效率，降低开发企业级应用程序的难度。

从实现原理看，容器与组件之间的通信除了 Java 程序本身的算法完成具体操作外，更重要的是通过一个部署描述文件来解决容器如何向组件提供服务，提供哪种服务的问题，这个文件是一个用 XML 语言写成的文件，称为部署描述文件。部署描述文件中详细描述了应用中的组件所要调用的容器服务的名称、参数等。部署描述文件就像组件与容器间达成的一个“契

约”，容器根据部署描述文件的内容为组件提供服务，组件根据部署文件中的内容来调用容器提供的服务。每个发布到服务器上的应用除了要包含自身实现的代码文件外，还要包括一个部署文件。

这个文件会随着系统的复杂而不断变得复杂，所以设计人员也在不断优化，比如 Java EE 5 推出了支持在组件中实现直接对注释的引用，取代配置复杂的部署描述文件。所谓注释，是 JDK 5 版本后支持的一种功能机制，它支持在 Java 组件的源代码中嵌入元数据信息，在部署或运行时应用服务器将根据这些元数据对组件进行相应的部署配置。容器在组件部署时通过提取注释信息来决定如何为组件提供服务。注释的出现大大简化了 Java EE 应用程序的开发和部署，是 Java EE 规范的一项重大进步。

进一步地，从 Java EE 6 规范开始，还引入了一种“惯例优于配置”或者称为“仅异常才配置”的思想，也就是对于 Java EE 组件的一些属性和行为，容器将按照一些约定惯例来自动进行配置，此时开发人员甚至连注释都可以省略。只有当组件的属性和行为不同于约定惯例时，才需要进行配置。这种编程方式大大降低了程序人员的工作量，也是需要开发人员逐渐熟悉和适应的一种编程技巧。

## 1.5　Java EE 容器及其服务

这一节介绍 Java EE 的容器以及容器提供的服务。

### 1.5.1　容器类型

Java EE 容器是组件和支持该组件的低级别、特定于平台的功能之间的接口。在 Web、企业 Bean 或应用程序客户端组件执行之前，它们必须组装到 Java EE 模块中，并部署到其容器中。有如下 4 种容器。

① Web 容器。Web 容器是 Web 组件和 Web 服务器之间的接口，管理 Java EE 应用程序中 Web 页面、servlet 和某些 EJB 的运行，给处于其中的组件提供一个环境，使得 JSP，Servlet 能直接和容器中的环境变量、接口交互而不必关注其他系统问题。Web 容器在 Java EE 服务器上运行。Web 组件可以是 Servlet 或 JavaServer Facelets 页面。

② EJB 容器。EJB 容器是 EJB 和 Java EE 服务器之间的接口，并管理 Java EE 应用程序中 EJB 的运行。只要满足 Java EE 规范的 EJB 放入该容器中，就会被容器进行高效率的管理，并且可以通过现成的接口来获得系统级别的服务，例如邮件服务、事务管理、安全、远程客户端的网络发布和资源管理等，而不用关心数据库连接速度、各种事务控制，这些都直接交给容器来完成。EJB 容器在 Java EE 服务器上运行。

③ 应用程序客户端容器。应用程序客户端容器是 Java EE 应用程序客户端与 Java EE 服务器之间的接口，管理应用客户端组件的运行。应用程序客户端及其容器运行在客户端上。

④ 小程序容器。管理小程序的运行。小程序容器包括一个 Web 浏览器和一个在客户端上运行的 Java 插件。

图 1-2 展示了 Java EE 服务器和容器。

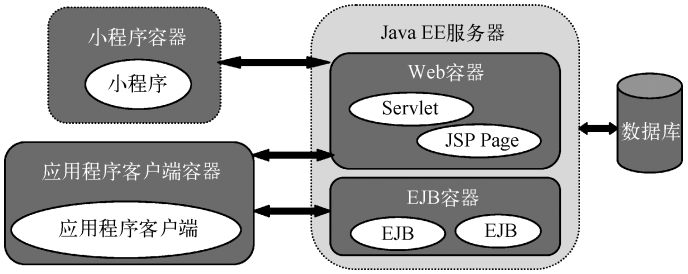


图 1-2 Java EE 服务器和容器

### 1.5.2 容器服务

Web 组件、EJB 组件或应用客户端组件运行前必须装配到一个 Java EE 模块中，并部署到它的容器中。装配过程包括为 Java EE 应用程序中的每个组件及 Java EE 应用程序本身进行容器设置。容器设置自定义 Java EE 服务器提供的底层支持，包括安全性、事务管理、Java 命名和目录接口（JNDI）API 查找及远程连接等服务。下面列出几种重要的服务。

① Java EE 安全（security）模型允许配置 Web 组件或企业 EJB，以使得只有被授权的用户才能访问系统资源。

② Java EE 事务（transaction）模型允许指定组成单个事务的方法之间的关系，以便将一个事务中的所有方法视为单个单元。

③ NDI 查找（JNDI lookup）服务为企业中的多个命名和目录服务提供统一接口，以便应用程序组件可以访问这些服务。

④ Java EE 远程连接（remote client connectivity）模型管理客户端和企业 Bean 之间的底层通信。创建企业 Bean 之后，客户端就好像和它在同一个虚拟机中一样调用其上的方法。

因为 Java EE 体系结构提供了可配置的服务，所以同一应用程序中的组件可以根据它们的部署位置表现出不同的行为。例如，EJB 可以具有安全设置，允许它在一个生产环境中对数据库数据进行一定级别的访问，并在另一个生产环境中对数据库进行一定级别的访问。

容器还管理不可配置的服务，例如 EJB 和 Servlet 生命周期、数据库连接资源池、数据持久性以及对 Java EE 平台 API 的访问。

## 1.6 Java EE 组件

Java EE 应用程序由组件组成。Java EE 组件是一个自包含的功能软件单元，它与相关的类和文件组装成 Java EE 应用程序，并与其他组件进行通信。Java EE 规范定义了以下 Java EE 组件。

① 运行在客户端的组件有应用客户端和小程序。

② 运行在服务器上的 Web 组件有 Servlet、JSF 和 JSP 组件。

③ 运行在服务器上的业务组件是 EJB 组件。

Java EE 组件是用 Java 编程语言编写的，其编译方式与使用 Java 语言编写的任何程序一样。Java EE 组件和“标准”Java 类之间的区别在于，Java EE 组件被组装到 Java EE 应用程序

中，它们被验证为格式良好且符合 Java EE 规范，并且被部署到生产环境中，由 Java EE 服务器运行和管理。

### 1.6.1　Java EE 客户端

Java EE 客户端通常是 Web 客户端或应用客户端。

**1．Web 客户端**

Web 客户端由以下两部分组成。

① 包含不同类型标记语言（HTML、XML 等）的动态 Web 页面，由运行在 Web 层的 Web 组件生成。

② Web 浏览器，呈现从服务器接收的页面。

Web 客户端有时称为瘦客户端（thin client）。瘦客户端通常不会查询数据库、执行复杂的业务逻辑或连接遗留应用程序。使用瘦客户端时，这些重量级的操作会转移到 Java EE 服务器上执行的企业 Bean，从而能提升 Java EE 服务器端技术在安全性、速度、服务和可靠性等方面的优势。

**2．应用程序客户端**

应用程序客户端（application client）在客户端主机上运行。有些任务要求有一个更丰富的用户界面，而标记语言无法满足这个要求，应用程序客户端就为用户提供了一种方法来处理这些任务。应用程序客户端通常有一个图形用户界面（Graphical User Interface，GUI），由 Swing API 或抽象窗口工具包（abstract window toolkit，AWT）API 创建，当然也可以有一个命令行界面。

应用程序客户端直接访问运行在业务层上的 EJB。不过，如果应用程序需求允许，应用客户端可以打开一个 HTTP 连接，与运行在 Web 层上的 Servlet 建立通信。用非 Java 的其他语言编写的应用客户端可以与 Java EE 服务器交互，使得 Java EE 平台具备与遗留系统、客户端以及非 Java 语言互操作的能力。

**3．小程序**

从 Web 层接收的 Web 页面可以包含嵌入的小程序。小程序是用 Java 编程语言编写的一个小客户端应用程序，在 Web 浏览器上安装的 Java 虚拟机中执行。不过，为了让小程序能够在 Web 浏览器中成功执行，客户端系统可能需要 Java 插件，还可能需要安全策略文件。

Web 组件是创建 Web 客户端程序的首选 API，因为客户端系统上无需插件或安全策略文件。另外，由于 Web 组件提供了一种方法可以将应用程序开发与网页设计分开，因此 Web 组件支持更简洁、更模块化的应用程序设计。所以，网页设计人员无须理解 Java 编程语言的语法也能很好地完成他们的工作。

**4．JavaBeans 组件架构**

服务器和客户端层也可以包含基于 JavaBeans 组件架构的组件（JavaBeans 组件），来管理以下元素之间的数据流：

① 应用客户端或小程序与 Java EE 服务器上运行的组件；

② 服务器组件与数据库。

根据 Java EE 规范，并不认为 JavaBeans 组件是 Java EE 组件。

JavaBeans 组件有属性，另外有访问这些属性的 get 和 set 方法。以这种方式使用的 JavaBeans 组件在设计和实现上通常都很简单，不过应当符合 JavaBeans 组件架构中指定的命名和设计约定。

**5．Java EE 服务器通信**

图 1-3 显示了构成客户端层的各个元素。客户端可以直接与运行在 Java EE 服务器上的业务层通信，如果客户端在浏览器中运行，还可以通过 Web 层中运行的 Web 页面或 Servlet 进行通信。

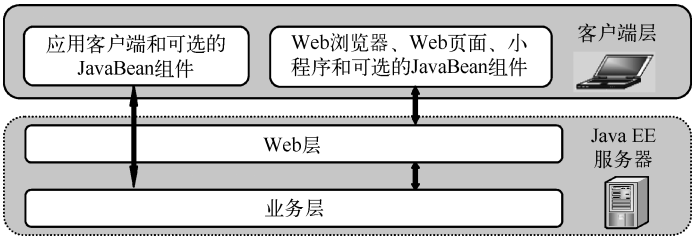


图 1-3　服务器通信

## 1.6.2　Web 组件

Java EE Web 组件是使用 JavaServerFaces 技术和/或 JSP 技术（JSP 页面）创建的 Servlet 或网页。Servlet 是 Java 编程语言编写的类，它会动态地处理请求和构造响应。JSP 页面是基于文本的文档，可以作为 Servlet 执行，不过允许采用一种更自然的方式创建静态内容。JSF 技术建立在 Servlet 和 JSP 技术之上，为 Web 应用提供一个用户界面组件框架。

装配应用程序时，静态 HTML 页面及小程序将与 Web 组件打包在一起。不过，根据 Java EE 规范，并不认为它们是 Web 组件。类似于 HTML 页面，服务器端的工具类也可以与 Web 组件打包在一起，同样也不认为它们是 Web 组件。

如图 1-4 所示，与客户端层类似，Web 层也可以包含一个 JavaBeans 组件来管理用户输入，并将该输入发送到业务层上运行的企业 Bean 进行处理。

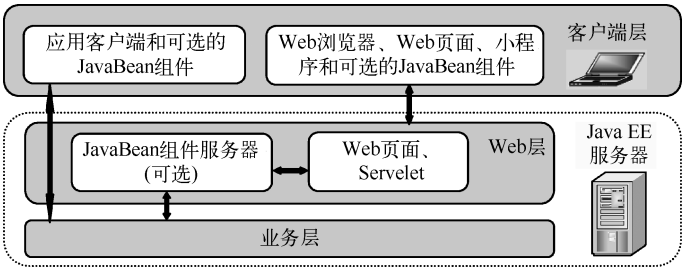


图 1-4　Web 层与 Java EE 应用

## 1.6.3　业务组件

业务代码是解决或满足一个特定业务领域（如银行、零售或金融）需求的逻辑，由业务

层或 Web 层上运行的企业 Bean 处理。图 1-5 展示了企业 Bean 如何接收来自客户端程序的数据，进行处理，并将其发送到企业信息系统层进行存储。企业 Bean 还会从数据存储中获取数据，进行处理，再将其发送回客户端程序。

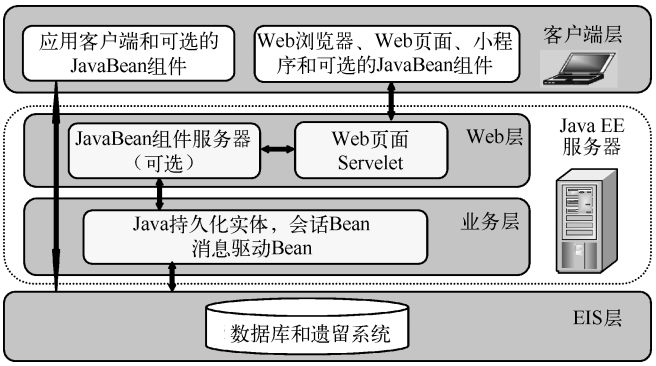


图 1-5　业务层和 EIS 层

## 1.7　Java EE 标准服务

Java EE 标准服务包括下述服务，一些标准服务实际上由 Java SE 提供。

### 1.7.1　HTTP

HTTP 客户端 API 在 java.net 包中被定义。HTTP 服务器端 API 由 Servlet、JSP、JSF 接口及 Web 服务支持来定义。

### 1.7.2　HTTPS

支持 HTTP 协议的客户端和服务器端 API 也同样支持带 SSL 协议上的 HTTP 协议。

### 1.7.3　Java™ 事务 API

Java 事务 API（Java transaction API，JTA）由两部分组成：

① 一个应用程序级的划分接口，容器和应用组件用它来划分事务。

② 一个介于事务管理器和资源管理器之间的 Java EE SPI 级接口。

### 1.7.4　RMI-IIOP

RMI-IIOP 子系统由 API 组成，这些 API 允许使用不依赖于底层协议的 RMI 风格编程，同样地，那些 API 的实现同时支持 Java SE 本地 RMI 协议和 CORBA IIOP 协议。通过支持 IIOP 协议，Java EE 应用就可以使用 RMI-IIOP 来访问 CORBA 服务，这些 CORBA 服务兼容 RMI 编程约束。这样的 CORBA 服务通常由 Java EE 产品之外的组件定义，通常是遗留系统。只要求 Java EE 应用程序客户端可以使用 RMI-IIOP API 来直接定义它们自己的 CORBA 服务。这样的 CORBA 对象通常用于在访问其他的 CORBA 对象时的回调。

正如在 EJB 规范中所述的一样，Java EE 产品必须使用 IIOP 协议输出企业 JavaBean 组件，并能访问企业使用 IIOP 协议的企业 Bean。使用 IIOP 协议使 Java EE 产品之间的交互成为可能，不过，Java EE 产品也可以使用其他的协议。当访问企业 Bean 时组件时需要使用 RMI-IIOP API 的要求在 EJB 3.0 中已经放开。

### 1.7.5 Java IDL

Java IDL 允许 Java EE 应用程序组件调用外部的使用 IIOP 协议 CORBA 对象。这些 CORBA 对象可以用任何语言编写，并且通常存在于 Java EE 产品之外。Java EE 应用程序可以使用 Java IDL 来作为 CORBA 服务的客户端，但是只有 Java EE 应用程序客户端可以直接使用 Java IDL 描述 CORBA 服务。

### 1.7.6 JDBC™ API

JDBC API 是用来连接关系数据库系统的 API。JDBC API 有两个部分：一个是应用程序组件，用来访问数据库的应用程序级接口，另一个是用于连接 JDBC 驱动和 Java EE 平台的服务提供者接口。Java EE 产品中对服务供应者接口的支持不是必需的。相反，JDBC 驱动应该被打包成一个资源适配器，该适配器使用连接器 API 机制来连接 Java EE 产品。JDBC API 包含在了 Java SE 中，但是本规范包含了对 JDBC 设备驱动的附加要求。

### 1.7.7 Java™ 持久化 API

Java 持久化 API（Java persistence API，JPA）是用于持久化和对象/关系映射管理的标准 API。该规范通过使用一个使用 Java 域模型来管理关系型数据库，为应用程序开发者提供了一种对象/关系映射机制。Java EE 必须支持 Java 持久化 API，也可以在 Java SE 环境使用中。

### 1.7.8 Java™ 消息服务

Java 消息服务（Java message service，JMS）是用于消息发送的标准 API，它支持可靠的“点对点”消息传递和“发布-订阅”模型。该规范要求 JMS 供应商同时实现“点对点”消息传递和“发布-订阅”型消息传递。Java EE 产品提供商必须提供一个预置的、默认的连接工厂供应用程序在访问 JMS 提供者时使用。

### 1.7.9 Java™ 命名和目录界面

JNDI（Java nancing and directory interface，JNDI）API 是用于命名和目录访问的标准 API。它有两个部分：一个是应用程序组件用来访问命名和目录服务的应用程序级接口，另一个是用于连接命名和目录服务供应商的服务供应商接口。JNDI API 包含在 Java SE 中，但该规范为它定义了附加标准。

### 1.7.10 JavaMail™

许多互联网应用程序都需要发送邮件的功能，因此 Java EE 平台包含了 JavaMail API 以及相应的 JavaMail 服务供应商 API，从而允许应用程序组件可以发送互联网邮件。JavaMail API

有两个部分：一个是应用程序组件用于发送邮件的应用程序级接口，另一个是 Java EE SPI 级的服务供应商接口。

### 1.7.11　JavaBeans™ 激活框架

JAF（JavaBeans activation francework，JAF）API 提供了一个框架来处理不同 MIME 类型的数据，它们源于不同的格式和位置。JavaMail API 使用了 JAF API。JAF API 包含在 Java SE 中，因此它可以被 Java EE 应用程序使用。

### 1.7.12　XML 处理

用于 XML 处理的 Java™ API（JAXP）支持工业标准化的 SAX 和 DOM API，用以解析 XML 文档，也支持 XSLT 转换引擎。XML 的流式 API（StAX API，Streaming API）为 XML 提供了一种拉模式解析型的 API。JAXP 和 StAX API 都包含在 Java SE 中，因此它们可以被 Java EE 应用程序使用。

### 1.7.13　Java EE 连接器体系结构

连接器体系结构是一种 Java EE SPI，它允许将支持 EIS 访问的资源适配器嵌入到任何 Java EE 产品中。连接器体系结构定义了一套标准的介于 Java EE 服务器和资源适配器之间的系统级协议集。这些标准协议包括如下几个方面。

① 连接管理协议。它让 Java EE 服务器池与底层 EIS 进行连接，并让应用程序组件连接到 EIS。这种方式构造出了一种可扩展的应用程序环境，用以支持大规模客户端对 EIS 系统的访问。

② 事务管理协议。该协议是事务管理器和 EIS 之间的一种协议，用以支持对 EIS 资源管理器的事务型访问。这个协议支持 Java EE 服务器使用事务管理器跨多个资源管理器来管理事务，也支持管理 EIS 资源管理器内部管理的事务，而不必涉及外部的事务管理器。

③ 对 EIS 进行安全访问的安全协议。这个协议为安全的应用程序环境提供支持，减少了对 EIS 的安全威胁，并保护了 EIS 管理的重要信息资源。

④ 线程管理协议。该协议允许资源适配器将工作委托给其他的线程，并允许应用程序服务器管理线程池。通过工作线程，资源适配器可以控制安全上下文和事务上下文。

⑤ 消息传递协议。它允许资源适配器将消息传递到消息驱动 Bean，独立于传递消息的特定消息形式、消息语义和消息传递基础架构。这个协议也充当标准消息提供方即插即用协议，它允许通过资源适配器将消息提供方插入到任何 Java EE 服务器中。

⑥ 允许资源适配器将导入的事务上下文传播到 Java EE 服务器的协议。该协议使得资源适配器与服务器和任务应用程序组件的交互成为导入事务的一部分，并保留导入事务的 ACID（即原子性、一致性、隔离性、耐久性）属性。

⑦ 可选协议。它提供了一个介于应用程序和资源适配器之间的通用命令接口。

### 1.7.14　安全服务

Java™ 认证与授权服务（Java authentication and authorization service，JAAS）使服务能够

基于用户进行验证和实施访问控制。它实现了标准的可插入式授权管理模块（pluggable authentication module，PAM）框架的一个 Java 技术版本，并支持基于用户的授权。容器的 Java™ 授权服务提供者协议（Java authorization contract for containers，JACC）定义了 Java EE 应用程序服务器和授权服务提供方之间的协议，允许将客户授权服务供应商插入任何 Java EE 产品中。容器的 JavaTM 授权服务提供商界面（Java authentication service provider interface，JASPIC）定义了一个 SPI，通过 SPI 实现消息授权机制的授权提供商可以集成到客户和服务器消息处理容器或者运行环境中。Java EE 安全 API 为使用 SPI 进行 Web 应用程序的用户授权提供了更便捷的途径，并定义了授权与认证的身份存储 API。

### 1.7.15 Web 服务

Java EE 为 Web 服务的客户端和终端提供了完整的支持。一些 Java 技术协同为 Web 服务提供支持。XML Web 服务的 Java API（Java API for XML-based web service，JAX-WS）和基于 XML 的 RPC 的 Java API（JAX-RPC，Java API for XML-based Remote Procedure Call）都为使用 SOAP/HTTP 协议的 Web 服务的调用提供了支持。Java SE 中的 JAX-WS 是支持 Web 服务的首选 API，它是 JAX-RPC 的继续。JAX-WS 提供了更广泛的 Web 服务功能，并提供对多重绑定/协议的支持。当使用绑定于 HTTP 协议的 SOAP1.1 协议时，JAX-WS 和 JAX-RPC 完全可以协同工作，但要受 WS-I 基本概要规范的约束。对 JAXR 的支持在 Java EE 7 中已经是可选项了。

JAX-WS 和 XML 绑定的 Java 体系结构（Java architecture for XML binding，JAXB）在 Java 类和 XML 之间定义了类似 SOAP 的映射，并且对 XML 模式提供了 100%的支持。使用 Java 附带的 API 的 SOAP（SOAP with attachments API for Java，SAAJ）对操作低层 SOAP 消息提供支持。Java EE 规范中的 Web 服务标准完整地定义了 Web 服务的客户端和终端在 Java EE 中的部署，以及使用企业 Bean 的 Web 服务终端的实现。Web 服务元数据规范定义了相应的 Java 语言注释来简化 Web 服务的开发。XML 注册 Java API（Java API for XML registries，JAXR）使客户端可以访问 XML 注册服务器。对 JAX-RPC 的支持在 Java EE 7 中已经是可选项了。

用于 JSON 处理的 Java API 提供了处理（语法分析、生成、转化和查询）JBSON 文本的便捷方法。JSON 绑定的 Java API 提供了 JSON 文本和 Java 对象之间便捷的转换方法。WebSocket 的 Java API 是用于产生 WebSocket 应用程序的标准 API。

RESTful Web 服务的 Java API（Java API for RESTful web services，JAX-RS）为使用 REST 风格的 Web 服务提供了支持。RESTful Web 服务更符合 Web 的设计风格，并且常常更易于使用多种编程语言对其进行访问。JAX-RS 提供了一个简单的高层 API 用于编写 Web 服务，以及一个低层 API 用于控制 Web 服务交互的细节。

### 1.7.16 并发工具

Java EE 的并发工具是标准的 API，通过如下对象类型提供 Java EE 应用程序组件的异步能力：管理执行服务、管理调度执行服务、管理线程工厂和上下文服务。

### 1.7.17 批处理

Java 平台 API（批处理）的批处理应用程序为批处理应用程序，调度和运行作业的运行时间提供了一个编程模式。

### 1.7.18　管理

Java 2 平台企业版管理规范中定义了一种 API，使用一种特殊的管理型企业 Bean 来管理 Java EE 服务器。Java™ 管理扩展（Java management extensions，JMX）API 也提供了一些管理上的支持。

### 1.7.19　部署

Java 2 平台企业版部署规范中定义了部署工具和 Java EE 产品之间的协议。Java EE 产品提供了运行在部署工具上的插入式组件，它允许部署工具将应用程序部署到 Java EE 产品中。部署工具提供了插入式组件可以使用的服务。对部署规范的支持在 Java EE 版本 7 中已经是可选项了。

## 1.8　企业级应用程序体系结构

应用程序体系结构是指应用程序内部各组件间的组织方式。企业级应用程序体系结构的设计经历了从两层结构到三层结构，再到多层结构的演变过程。

### 1.8.1　C/S 两层结构

C/S 应用程序两层体系结构模式如图 1-6 所示。在两层体系结构中，客户层的客户端应用程序负责实现人机交互、应用逻辑、数据访问等职能；服务器层由数据库服务器来实现，主要职能是提供数据库服务。

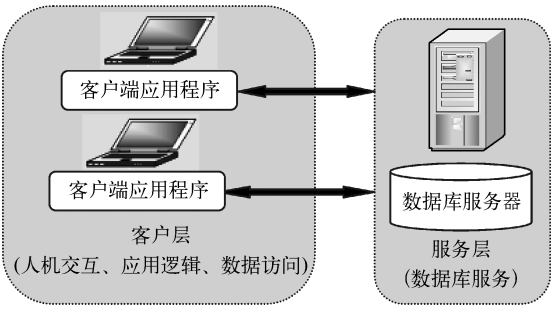


图 1-6　C/S 两层结构示意图

这种体系结构的应用程序有以下的缺点。

① 安全性低。客户端程序与数据库服务器直接连接，非法用户容易通过客户端程序侵入数据库，造成数据损失。

② 部署困难。集中在客户端的应用逻辑导致客户端程序肥大，而且随着业务规则的不断变化，需要不断更新客户端程序，大大增加了程序部署工作量。

③ 耗费系统资源。每个客户端程序都要且接连到数据库服务器，使服务器为每个客户端建立连接而消耗大量宝贵的服务器资源，导致系统性能下降。

### 1.8.2　B/S 三层结构

为解决两层体系结构应用程序带来的问题，人们又在两层体系结构应用程序的客户层与服务器层之间又添加了一个第 3 层应用服务器层，提出三层体系结构应用程序，即客户层、应用服务器层、数据服务器层，如图 1-7 所示。

客户层应用程序通常由浏览器（browser）程序实现，因此这种体系结构又被称作 B/S 模式。与两层体系结构的应用相比，三层体系结构应用程序的客户层功能大大减弱，只用来实现人机交互，原来由客户层实现的应用逻辑、数据访问职能都迁移到应用服务器层上来实现。数据服务层仍然仅提供数据信息服务。由于应用服务器层是位于客户层与数据服务器层中间

的一层，因此应用服务器被称作“中间件服务器”或“中间件”，应用服务器层又被称作“中间件服务器层”。

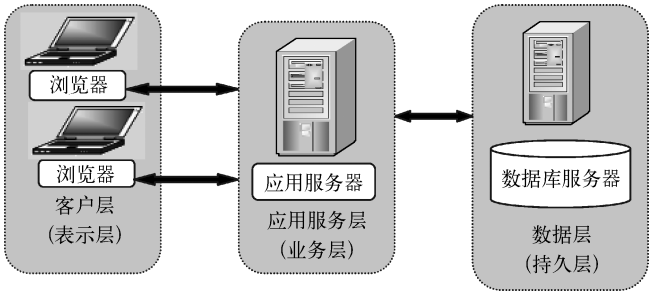


图 1-7　B/S 三层结构示意图

相对于两层体系结构的应用程序，三层体系结构的应用程序具有以下优点。

① 安全性高。中间件服务器层隔离了客户端程序对数据服务器的直接访问，可保护数据信息的安全。

② 易维护。由于业务逻辑在中间件服务器上，当业务规则变化后，客户端程序基本不做改动，只需要升级应用服务器层的程序即可。

③ 快速响应。通过中间件服务器层的负载均衡及缓存数据能力，可以大大提高对客户端的响应速度。

④ 系统扩展灵活。基于三层分布体系的应用系统，可以通过在应用服务器部署新的程序组件来扩展系统规模；当系统性能降低时，可以在中间件服务器层部署更多的应用服务器来提升系统性能，缩短客户端的响应时间。

可以将中间件服务器层按照应用逻辑进一步划分为若干个子层，这样就形成了多层体系结构的应用程序。关于多层体系结构应用程序，在有些文献中也将二层及三层以上体系结构应用程序统称为多层体系结构应用程序。

### 1.8.3　多层结构

Java EE 将企业应用程序划分为多个不同的层，并在每一层上定义对应的组件来实现它。典型的 Java EE 结构的应用程序包括四层，分别是客户层、表示层（Web 层）、业务逻辑层和企业信息系统层，如图 1-8 所示。

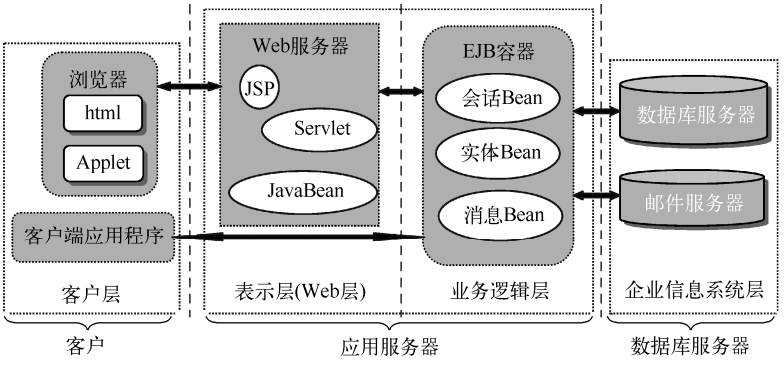


图 1-8　四层结构示意图

客户层可以是网络浏览器或者桌面应用程序。表示层（Web 层）、业务逻辑层都位于应用服务器上，它们是由 Java EE 标准组件 JSP、Servlet、EJB 和 Entity 等来实现的，这些组件运行在实现了 Java EE 标准的应用服务器上，以实现特定的表示逻辑和业务逻辑。企业信息系统层主要用于企业信息的存储管理，主要包括数据库系统、电子邮件系统、目录服务系统等。Java EE 应用程序组件经常需要访问企业信息系统层来获取所需的数据信息。更具体地叙述如下。

① 客户层：包括浏览器（HTML 或小程序）和桌面应用程序。

② 表示层（Web 层）：包括 Servlet，JavaBean 和 JSP。

③ 业务逻辑层：EJB。

④ 企业信息系统层：如 Database、ERP、大型机事务处理和其他遗留信息系统。

### 1.8.4　Java EE 的分层模型与框架

框架（framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法。另一种定义认为，框架是可被应用开发者定制的应用构架。前者是从应用方面给出的定义，后者是从目的方面给出的定义。

框架要解决的最重要的一个问题是技术整合的问题，在 Java EE 的框架中，有着各种各样的技术，不同的软件企业需要从 Java EE 中选择不同的技术，这就使得软件企业最终的应用程序依赖于这些技术。技术自身的复杂性和技术的风险性将会直接对应用造成冲击。而应用程序是软件企业的核心，是竞争力的关键所在，因此应该将应用程序自身的设计和具体的实现技术松绑。这样，软件企业的研发将集中在应用的设计上，而不是具体的技术实现，技术实现是应用的底层支撑，它不应该直接对应用产生影响。框架一般处在低层应用平台和高层业务逻辑之间的中间层。

开发架构一般都是基于两种形式，一种是 C/S 架构，另一种是 B/S 架构。在 Java EE 开发中，几乎全都是基于 B/S 架构的开发。那么在 B/S 架构中，系统标准的三层架构包括：表现层、业务层、持久层，如图 1-7 所示。三层架构中，每一层各司其职。

① 表现层：接收请求展示数据。

也就是常说的 Web 层。它负责接收客户端请求，向客户端响应结果，通常客户端使用 HTTP 协议请求 Web 层，Web 需要接收 HTTP 请求，完成 HTTP 响应。 表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。表现层的设计一般都使用 MVC 模型（MVC 是表现层的设计模型，和其他层没有关系）。

② 业务层：处理业务逻辑。

也就是常说的 Service 层。它负责业务逻辑处理，和开发项目的需求息息相关。Web 层依赖业务层，但是业务层不依赖 Web 层。业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性，也就是常说的，事务应该放到业务层来控制。

③ 持久层：与数据库交互。

也就是常说的 DAO 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗地讲，持久层就是和数据库交互，对数据库表进行增删改查。

一些企业根据这个分层，开发了一些框架，来实现各层的功能。例如，表现层的 Spring

MVC 框架，Struts 1 框架和 Struts 2 框架；业务逻辑层的 Spring 框架；和持久层的 Hibernate 框架和 MyBatis 框架。这些产品也都满足 Java EE 的标准。

在实际的开发中，往往根据项目开发的实际来选择框架，这样就形成了不同的框架组合形式。比较常见有两种组合形式：一种是使用 Struts2 框架实现表现层，使用 Spring 框架实现业务层（Service），使用 hibernate 实现实现持久层（DAO），这种组合形式称为 SSH 模式；第二种就是使用 Spring MVC 框架表现层，使用 Spring 框架实现业务层，使用 MyBatis 框架实现持久层（DAO），这种形式称为 SSM 模式。本书选择第二种组合形式进行讲解。各层和对应框架如图 1-9 所示。

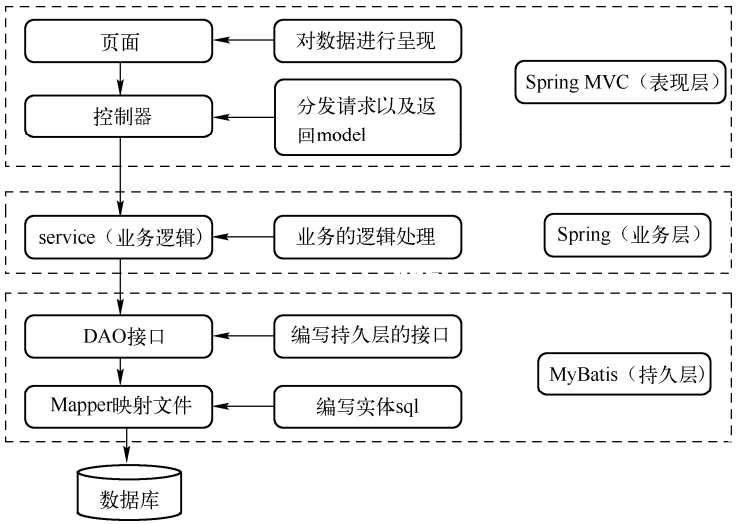


图 1-9 分层模型与框架

## 1.9 集成环境及配置简介

为了便于学习和程序调试，本节简单介绍一下本书使用的开发环境及环境配置。

### 1.9.1 集成环境简介

本书使用的集成环境是 Eclipse 2020-06 版本，Tommcat 使用的是 8.5 版，MySQL 使用的是 8.0 版，账户是“root”，密码是“111111”。为了正确处理本书中涉及的λ表达式，JDK 要用 1.8 及其以上版本，这里使用的是 11 版。Maven 依赖的包，其版本在 1.9.2 中给出的 pom.xml 文件中给出了。在建立 Maven 工程后，只要导入该文件，则这些 jar 包就自动下载更新了。

在上述环境和配置下，本书所有程序都通过了调试，能正常运行。图 1-10 是 Eclipse 集成环境的界面。

另外，三个地方的 JDK 版本最好保持一致。下面进行说明。

① 选择工程 myhomework，在该工程上单击鼠标右键，弹出如图 1-11 所示的界面。

在如图 1-11 的界面中选择 Properties 属性，得到如图 1-12 所示的界面。在该界面中选择左侧列中的 Project Facets 属性，再选择中间列的 Java 属性，在右侧列中选择 Java 的版本，使得修改后的 Java 版本保持一致，如图 1-12 所示。

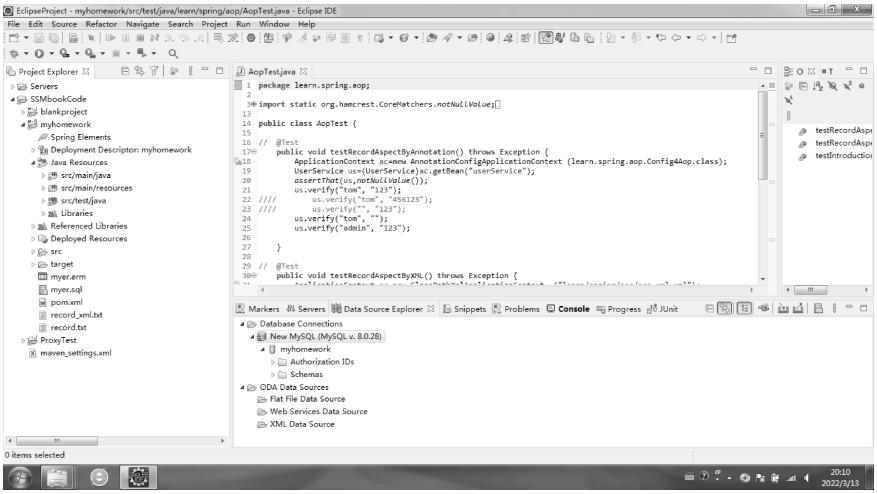

图 1-10　Eclipse 集成环境和本书的程序展示

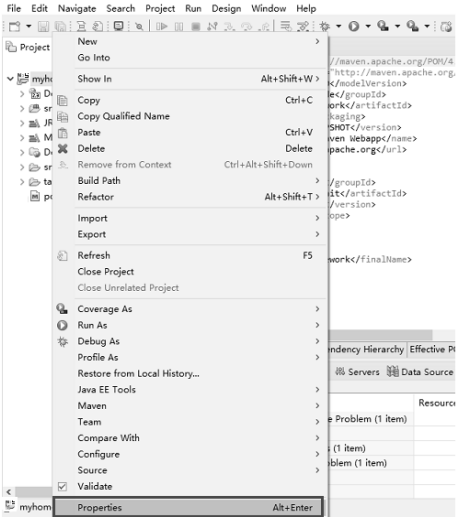


图 1-11　设置 Properties 属性

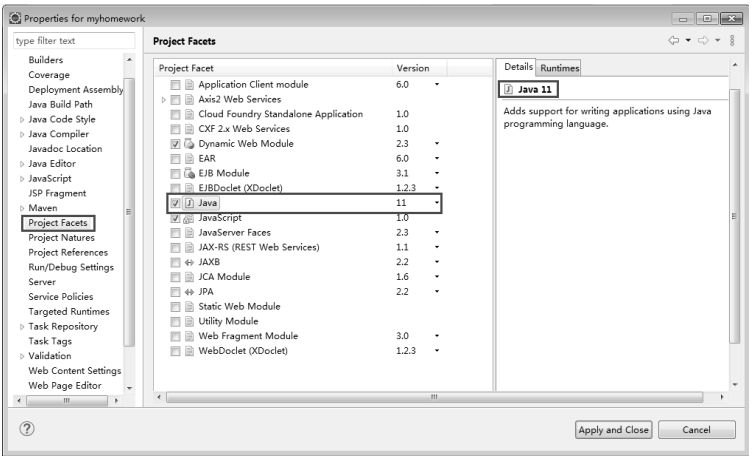


图 1-12　设置 Project Facets 属性

② 在如图 1-10 的界面中，选择菜单栏中的 Window 项，得到如图 1-13 所示的界面。

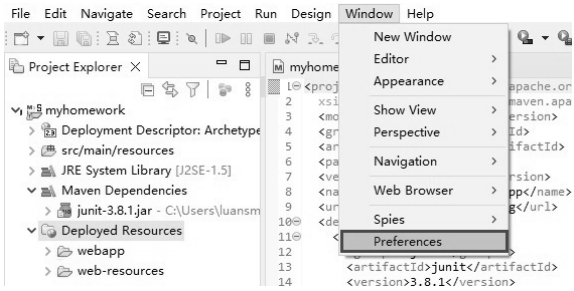


图 1-13　设置 Preferences 属性

在如图 1-13 所示的界面中选择 Preferences 属性，得到如图 1-14 所示的界面。单击左侧栏中的 Java 项，选择其中的 Compiler 属性，在右侧栏中设置 Compiler compliance level，使得版本一致，这里选择的是“11”。

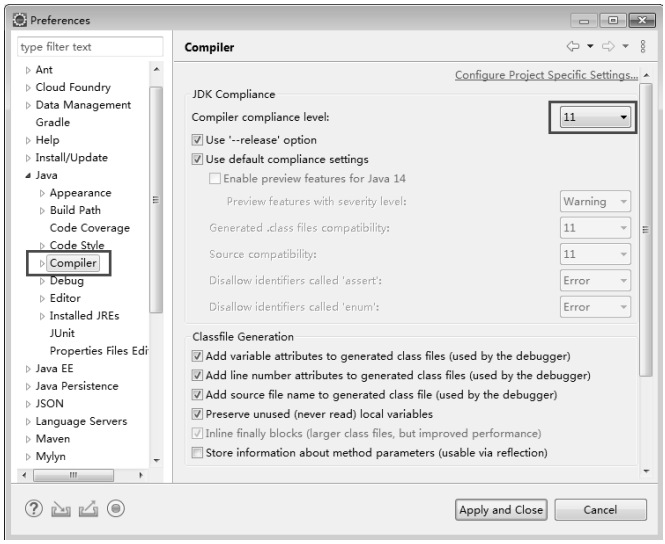


图 1-14　设置 Compiler 属性

③ 在如图 1-14 的界面中选择 Installed JREs 属性，设置或者选择右侧的 Installed JREs，使得版本一致，这里选择的是 JDK11，如图 1-15 所示。

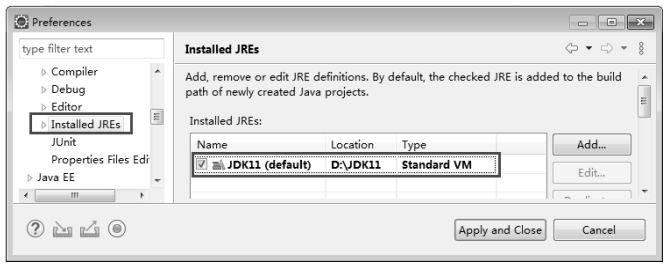


图 1-15　设置 Installed JREs 属性

如果使用 Eclipse 2021 较新的一些版本，需要在 Maven 工程的 pom.xml 文件中，在<build>模块中增加如下代码，以升级 Maven 插件版本，否则有错误提示。

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.1</version>
    </plugin>
</plugins>
```

构建 Maven 工程的时候，初始 JRE 是 1.5 版的，替换为更高的版本后，Maven 工程只要更新后，JRE 就会变回 1.5。这时在 pom.xml 文件的<build>模块中增加更新插件即可。例如，要用 11 版本，可以增加如下代码。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <source>1.11</source>
        <target>1.11</target>
    </configuration>
</plugin>
```

### 1.9.2　环境配置

这里展示了 Maven 工程的设置内容，在建立自己的 Maven 工程后，把下面的内容复制到 pom.xml 文件中，就自动下载更新 Maven 工程所依赖的包，这也是本书实例使用的依赖包。

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-core</artifactId>
    <version>1.3.RC2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>1.3.RC2</version>
```

```
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.19</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.5</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.2.59</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
    <dependency>
        <groupId> org.aspectj</groupId>
        <artifactId> aspectjweaver</artifactId>
        <version> 1.8.7</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.5</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-juli</artifactId>
        <version>9.0.39</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
```

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.8.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.8.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-web -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.2.8.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<!--  使用 Json 所依赖的 jar 包  -->
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.9.8</version>
</dependency>
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.8</version>
</dependency>
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.9.8</version>
</dependency>
<!--  读写 Office 文档所依赖的 jar 包  -->
<dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.16</version>
</dependency>
<dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
       <version>2.0</version>
</dependency>
```

### 1.9.3 关于测试

在程序设计的过程中，完成一个模块后就可以进行测试，本书使用的单元测试工具是junit 4.12。该包已经在 Maven 的 pom.xml 文件中给予了设置，会自动下载更新。

在完成一个模块后，就可构建一个测试类来完成测试。测试类一般都放在路径 src/test/java 下，如图 1-16 所示。2.6.4 节中的“4. 编写测试用例”详细说明了导入 junit 4.12 的方法，并说明了构造测试类进行测试的过程。本书后面的部分将省略这个步骤，只列出测试代码，读者自己按照测试步骤处理即可。

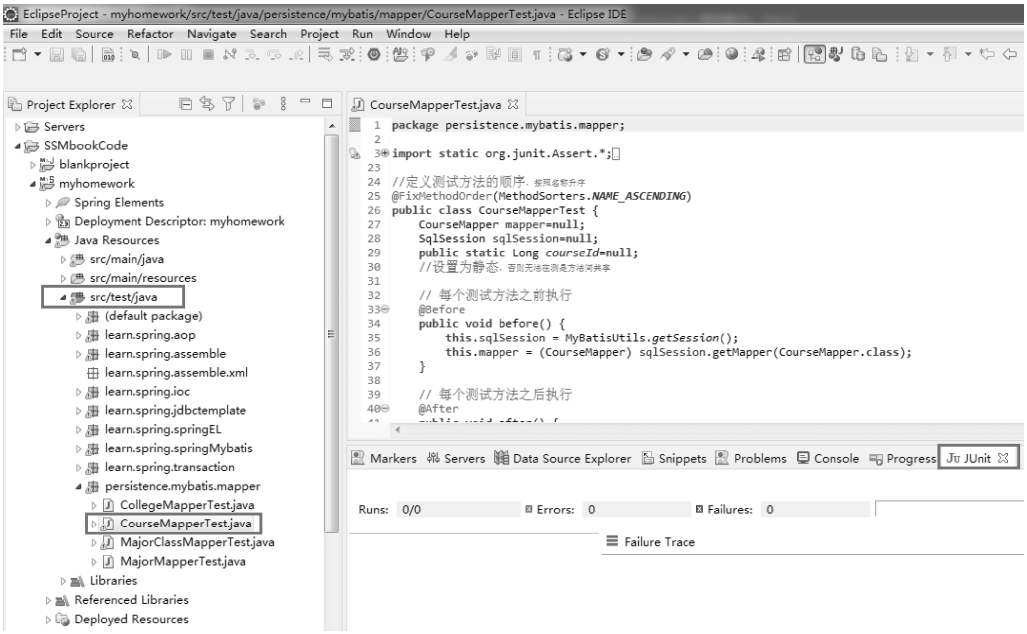


图 1-16 测试类实例展示

## 1.10 本章小结

本章简单介绍了 Java EE 的产生与发展，平台的构成，以及容器、组件等基本概念。通过本章的学习，掌握 Java EE 平台的组成及体系结构；企业级应用程序开发的思想和方法，包括容器/组件的编程思想，分层实现的应用程序模型；以及 Java EE 平台提供的服务。掌握 Maven 依赖导入的方法和测试类的建立和使用。

## 习题

1. 简述 Java EE 产生的背景及其思想。
2. 简述 Java EE 体系结构的主要技术。
3. 简述 Java EE 的 API。