J2EE 交流文档

概述

曾有人说过：程序等于算法加数据结构。但是在今天，这句话已不适用大部分的商业应用开发。我们面临一个需求容易变化、业务逻辑复杂、多数据源、多表现形式的应用情形。如何快速、高质量地完成项目开发，如何避免公司内部多项目组的重复劳动，建立灵活而强大的应用架构以提高整体生产力，这是目前很多开发人员和公司管理层面临的难题。本文从实用主义出发，避免纯技术的讨论，对实际项目中一些热点难点进行讨论，力求给读者一些启示。限于笔者的经验和能力，其中不足和错误之处还望指正。

第一部分是对若干专题的讨论以及优秀开源架构 SpringFramework 技术的介绍，第二部分是一个实际的项目的设计和开发。附录 A 是一份设计文档，是对第一部分数据采集部分的补充。

注：这篇文档是笔者学习和使用Java到现在的一点总结，旨在和cjsdn的网友分享经验与教训。部分内容因涉及具体工作而比较简略，敬请见谅。

目录

第一部分 专题讨论 2
数据库编程 2
避免前台的或核心的代码直接访问数据库。 2
DAO模式 3
不应过分强调应用的数据库移植性 4
适当地使用O/R Mapping技术简化数据库的读写 5
实例 5
ORM 实现技术 5
Hibernate的简单介绍 6
底层的JDBC技术往往更有效 6
SpringFramework JDBC Support 7
实例 8
Spring对SQLException的封装 9
runtime exception vs. checked exception 9
多数据源 10
Web层 11
MVC的Web开发方式 11
SpringFramework WebMVC 12
SpringFramework的简单介绍 15
Goal of SpringFramework 15
利用面向对象的优秀技术 15
将面对接口编程做到实处 16
将单元测试变的简单 16
SpringFramework的概貌 16
BeanFactory和IoC容器 17
什么是BeanFactory？ 17
什么是IoC? 19

Spring技术构架 20
Power the whole application with Spring 20
Integrate with third-party web application frameworks 21
Use Spring to manage EJBs 22
Remoting through SOAP, RMI, Web services 22
参考资料 23
第二部分 Demo系统 23
附录A 23

第一部分 专题讨论

J2EE，开发企业应用很好的一项技术或者说平台。它的内涵非常丰富，使用人群也非常庞大，有很多最佳实践，优秀的工具，优秀的 api 和 framework。事实证明，J2EE 确实可以为复杂的企业应用提供强大的技术保障。但或许由于它过于复杂，开发人员缺少足够的技能或开发的经验，往往也是导致一些 J2EE 项目落马的原因。
这一部分中，我花一定篇幅谈谈学习和使用 J2EE 技术到现在对若干问题的看法。大部分内容都不会讲的很细节，因为我假设读者具备相当的这方面知识。

数据库编程

在今天，几乎每个应用都会使用数据库来保存信息。这方面我们遇到的挑战是：

- 如果需求明确，部署情形也比较单一，则往往针对一个特定数据库产品的某一版本设计 schema。应用的复杂性往往导致会建立很多表，很多表与表之间的联系，还有些诸如字段的约束，触发器、存储过程等数据库逻辑。
- 有时，一个项目需要同时使用几个数据库，或是需要把项目部署在不同的数据库中。

如何处理好数据持久层..我的看法和建议是：

避免前台的或核心的代码直接访问数据库。

把数据库读写的代码分散在应用系统的各个地方，这是一种冒险的做法。数据库 schema 的信息被扩散到应用的各个角落，万一 schema 发生一点变化，如字段名发生变化或一个字段从一个表转移到另一个表，就会牵扯很多代码。另外编写高质量的数据库读写代码也是需要一定技能的，比如如何恰当地提交事务或回滚，如何在错误发生后回收资源。举个例子，这样的代码或许很多人都看到过：

```
try{
 Connection con=...//getConnection
 Statement stat=con.createStatement("select * from tab");
 ResulstSet rs=stat.executeQuery();
 ..//处理结果集
```

```
 rs.close();
 stat.close();
 con.close();
}catch(SQLException e){..}
```

这里如果 rs.close()处发生异常，conneciton 将不能被关闭，造成资源的浪费

往往有些项目组中选择自己开发了数据库读写 api，如

```
DbClass db=new DbClass();
RecordSet rs=db.query("select * from tab");
```

这个模块一般由比较熟悉数据库编程的开发人员负责设计和开发。但是遗憾的是，我很少看到编得比较出色的这种实用类——往往是性能不佳，灵活度不大甚至犯了类似前面的低级错误。

其实目前有些开源项目(OpenSource Project)提供了比较高质量的类似功能的实用类或框架，直接使用它们我认为是更好的选择。

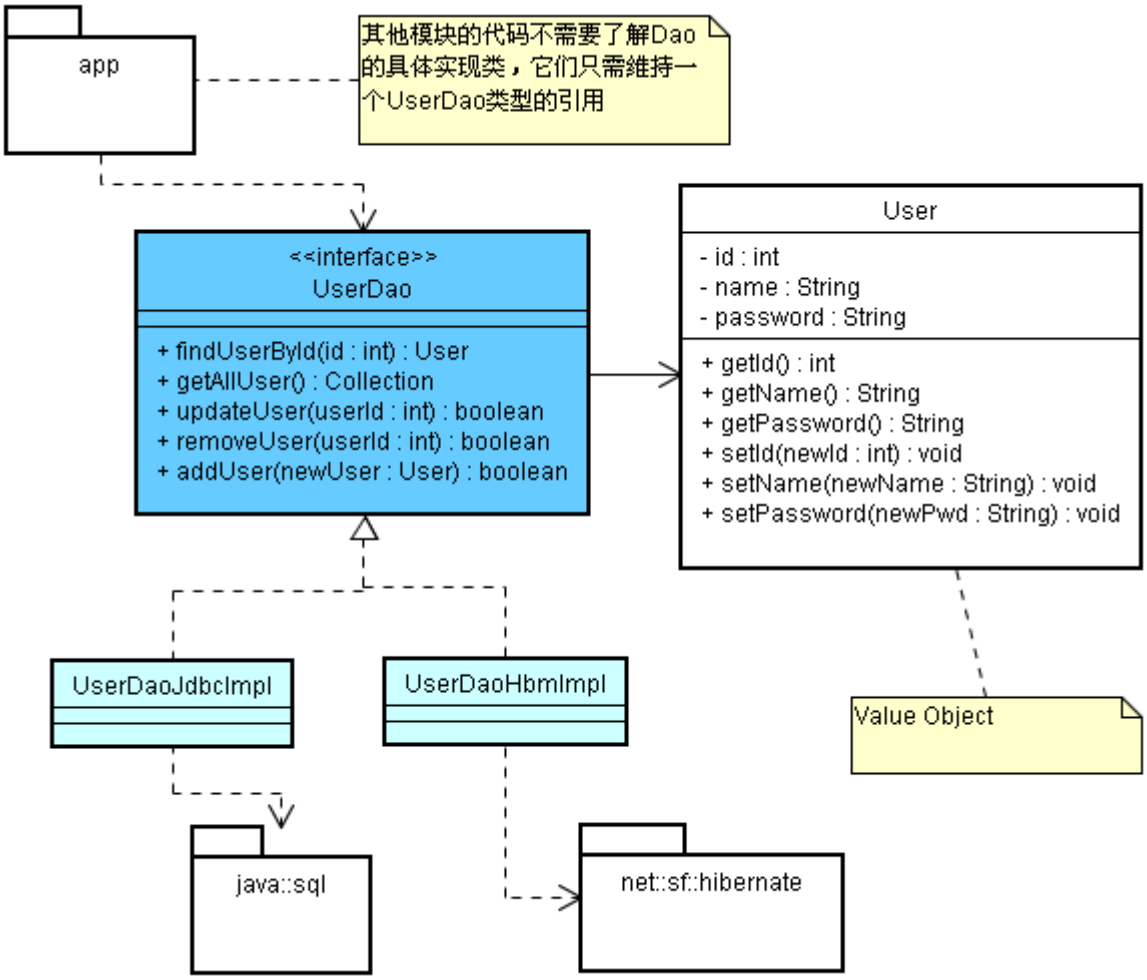
http://jakarta.apache.org/commons/dbutils

http://www.springframework.org/

DAO 模式

为了避免数据库读写的代码充斥在应用程序中，我们可以使用 DAO 模式，集中进行数据的读写。DAO(Data Access Object)是一个基本的 J2EE 设计模式，往往配合 VO(Value Object)来简化数据的读写和传递。

典型的，如果我们要处理数据库的 user 表，可以建立一个 User 类，描述了 user 表的数据。然后建立一个接口 UserDao，规定系统关于 User 操作的接口。系统的其他模块都只和这个接口打交道，而无需知道实现类。我们可以选用一种或多种技术来实现这个接口——可以是基于 JDBC 或是 Hibernate,JDO 等；甚至可以把基于 JDBC 的 DAO 实现类也分为面向 MySQL 或是面向 Oracle 的。这样，变动一项具体的数据库访问技术或是更改了数据库产品，只需要在实现类中做改动，而不需要更改应用的其他代码。对应的 UML Class Diagram 如下:



另外，DAO 模式不光可以解决多数据库和数据库移植的问题，他同样适用于非数据库形式的数据源。在下文“多数据源”专题中会讨论这个。

不应过分强调应用的数据库移植性

Java 具有先天的优势，JDBC 很好地抽取了数据库的共性。在实际项目中，确实会发生从一个数据库转到另一个数据库的情形。因而有不少人抱有这样的观点：Java/J2EE 项目应着重考虑数据库的可移植性，尽量少的使用当前数据库的特性，如自增字段、sequence、trigger、stored procedure 等。

这种观点不无道理，但是我认为在大多数情形下，这样做是无益的。如果系统已经比较明确的使用一种特定数据库（如 Oracle），公司花费很多财力来购买 license, 聘请 DBA，如果我们只使用它很少的功能，以追求所谓的数据库移植性，某种程度上这无疑是一种浪费。

一般来说数据库只保存数据，应用逻辑应该分离出来，在应用代码中完成。但是有些操作，直接使用数据库的 trigger,store procedure 来完成或许要比由应用程序来完成高效的多。即使我们需要面对多种数据库，我们也还是可以使用数据库的特性来提高效率，以提高公司资源的利用度。毕竟主流的数据库在功能度方面是类似的，只需要做一定的包装，就不会妨碍我们使用特定数据库的增强特性。

适当地使用 O/R Mapping 技术简化数据库的读写

由于我们借助 Java 技术用面向对象思想开发项目，所以试图用面向对象的方式来处理数据库是很自然的。 数据库提供的是二维的接口，而按对象方式组织数据可以有一定的关系和层次。ORM（Object/Relative Mapping）就是提供了这种印射关系。它的好处是不言而喻的——我们可以统一地使用面向对象思想来编写业务逻辑和数据处理模块。

实例

举个例子，原先我们如果要更改一个用户名，代码或许会这样实现

```
boolean renameUsername(int userId,String newName)...
  String sql="update user set name=? where id=?";
  Connection conn=getConnectin();
  PreparedStatement pStat=conn.prepareStatement(sql);
  pStat.setString(1,newName);
  pStat.setInt(2,userId);
  int row=pStat.executeUpdate();
  closeSafe（pStat,conn）;
  return row==1;
}
```

而如果使用 ORM 技术，或许只需要

```
User user=session.get(User.class,new Integer(userId));
user.setName(newName);
session.update(user);
```

更明显的优势是：对于数据库多表关联，我们一般需要编写带有 JOIN 的 SQL 语句，而使用 ORM 技术可以透明的处理这种多表关联。

如语句 `query("from User user where user.commpany.name='foo' and user.age>30")` 可以找出 30 岁以上，在 foo 公司任职的人（数据库中 user 表有个字段是 company_id,是表 company 的外键）。当然底层实现是由 ORM 工具动态生成 SQL 语句，但这样比我们手动写要方便和保险的多。

ORM 实现技术

ORM技术在近些年是应用的热点，出现了很多优秀的实现技术，其中J2EE的Entity Bean尤其是CMP就是一种ORM尝试——它使用EJB-QL来实现查询，项目中通过Entity Bean而不是直接通过JDBC访问数据库来读写数据。另外两种主流的ORM技术是JDO和Hibernate。前者是Sun公司JCP下制定的规范，1.0 已推出 2 年了，更强大的 2.0 规范正在制定中。JDO的优势是，它作为规范可以由不同厂家实现不同的产品，而保持接口的一致。后者Hibernate可能用的更广一点，它是开源的项目(http://www.hibernate.org)，并不遵循JDO规范，以自己的方式实现了ORM，包括提供了强大而简易的查询语言HQL(Hibernate Query Language)，也提供类似JDO的Criterion方式的查询api。

Hibernate 的简单介绍

Hibernate 的发行包中提供了比较丰富的文档和范例，应该说上手是比较容易的。我的习惯做法是，先设计数据库表，然后使用工具读取表信息生成 Java 类文件和 mapping 文件，然后手动做一些修改如添加主键生成算法、一对多、多对多关系等。另外，也尝试过使用 XDoclet 和 Ant, 在 Java 类文件中用注释的方式提供 mapping 信息，由 XDoclet 来生成 hbm.xml 文件。
Hibernate 本身提供一些工具可以由类生成数据库定义(DDL)和 mapping, 但我不倾向这么做。设计数据库的工作还是由专业的 DBA 来完成比较好，借助他们对性能优化和设计上的一些经验。另外往往一个数据库同时为几个应用系统服务，有些是 Java 写的，有些不是。甚至一个数据库的寿命要比 Java/J2EE 项目的还长。以上种种，让我认为先有表再生成 Java 类更合情合理些。

Hibernate 比较注重使用当前数据库的特性。以一个分页功能为例，使用 Hibernate 的实现是

```
Query q = session.createQuery("from News as news where ...");
q.setFirstResult(11);
q.setMaxResults(10);
List l = q.list();
```

Hibernate 会尝试使用当前数据库对分页的支持来完成这一功能，如 Oracle 提供的 rownum, MSSQL 和 DB2 提供的 TOP, MySQL 和 PostgreSQL 提供的 LIMIT 字句。如果数据库本身不支持分页，则会使用 JDBC 2 的可滚动结果集，来绝对定位到起始记录来得到一定条数的纪录。由于 JDBC 中的 ResultSet 是保持连接的，并非一次性把数据全部读到本地，绝对定位会跳过前面的纪录，所以性能也不差。如果当前数据库既不支持分页 SQL 也没有提供 JDBC2 以上的驱动，则会使用 JDBC1 的方式，用 resultSet.next()来跳过若干纪录，来实现读取特定区域的纪录

Hibernate 在延迟读入、大对象处理、多表关联等方面都做得比较灵活和强大。难怪它的支持者人数要比 JDO 的多一些，成为当前最主流的 O/R Mapping 技术之一。

尽管 Hibernate 等 ORM 技术大大提高了我们的生产力，并使数据库持久层的代码和业务逻辑层一样，符合面向对象的思想。但我们还是应该谨慎选择和使用 ORM 技术，并且尽量小范围地使用这些 API（如仅在 DAO 实现类中），不要让系统受限于当前使用的数据持久技术。虽然很多 ORM 技术都提供本地的高速缓存和对象延迟加载来提高性能，但是有些时候直接使用 JDBC 或许效率更高。

另外，由于 Hibernate 比较好地做到了适应各个数据库，所以对提高应用适应多数据源很有帮助。以下文的 Demo 系统为例，笔者最初使用 Linux 上的 PostgreSQL，后来转到 MySQL，最后使用 hsqldb。这几次变迁代码更改的很少，基本只对 Mapping 文件和 Hibernate 的配置参数做了少量修改。

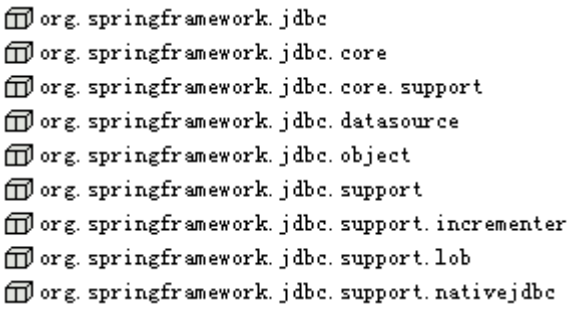
底层的 JDBC 技术往往更有效

正如前面说的，虽然现在有比较优秀的 ORM 技术供我们选择，但并不表示 ORM 优于传统的 JDBC 技术，反之亦然。JDBC 提供了非常直接也非常有效的方式来和数据库交互。有经验的数据库开发人员可以编写出效率极高的 SQL 语句来完成某项工作，而如果用 ORM 或是朴素的 SQL+JDBC 来做，或许代码量要高一个数量级。
不过，也正如本节一开始所说的，JDBC 是个比较底层的 api，编出高质量的 JDBC 代码，对程序员也是不小的考验。我个人比较推荐使用 Springframework 的 JDBC 模块和 Jakarta commons DbUtils 等的

成熟 API 来简化我们的工作.

SpringFramework JDBC Support

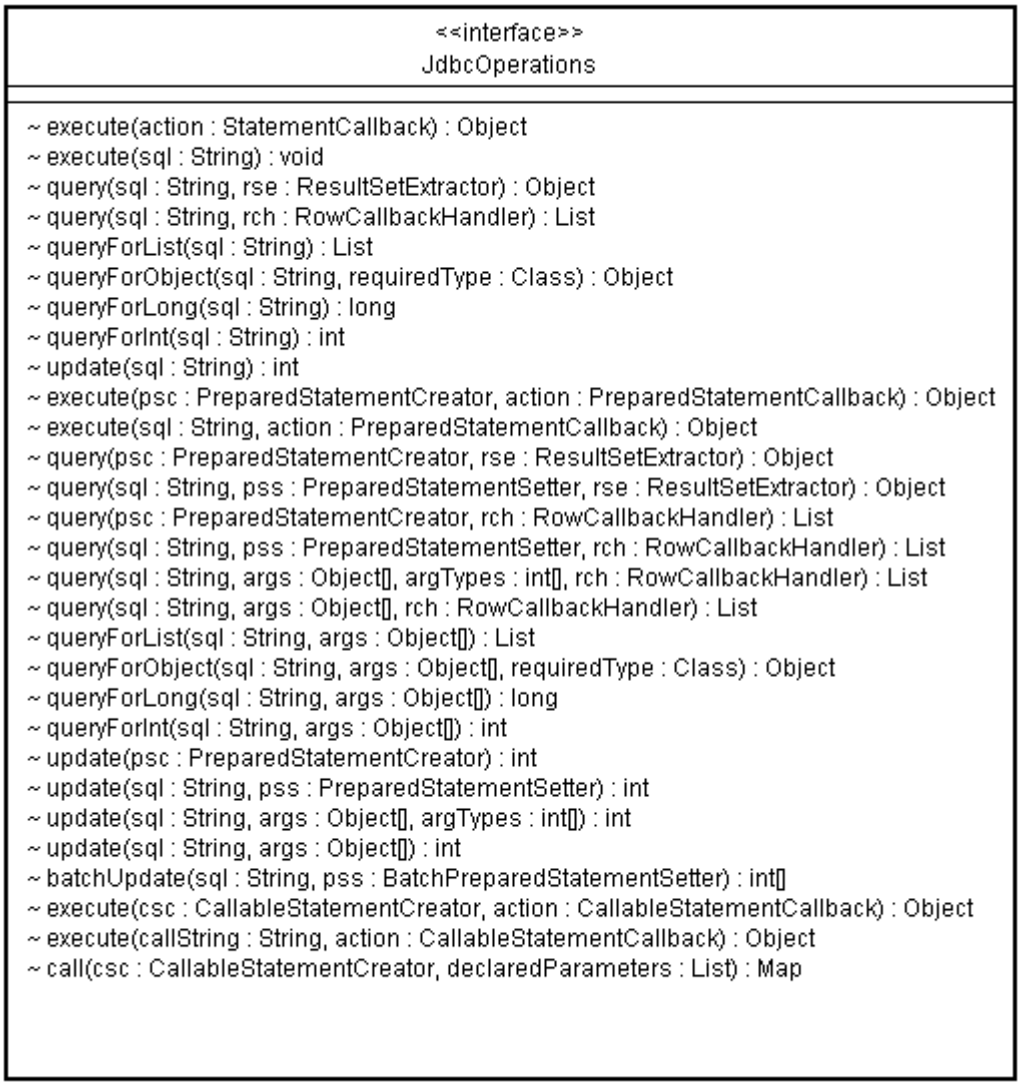
相关的包如下：



和 Spring 的其它模块一样，jdbc support 使用 IoC 模式来分离数据库资源。(IoC 模式将在下文详细介绍)

使用时首先新建一个 org.springframework.jdbc.core.JdbcTemplate 对象，调用它的 `setDataSource(javax.sql.DataSource dataSource)` 方法设置数据源。一般我们使用 JNDI 来获取容器中的数据源，在没有容器的情况下可以使用类似 Jakarta Commons DBCP 这样的本地连接池来建立数据源。

数据读写需要调用对应的 query 或 execute 方法。JdbcTemplate 主要实现了 JdbcOperations 方法，提供简单到复杂的数据读写方法。



实例

下面举几个例子，演示它的用法
数据表　user(id,name,age)　id 为 int 型主键，name 为 char(10), age 为 int 型

- 查询指定id用户的年纪

```
int getAge(int userId){
   return jdbcTemplate.queryForInt("select age from user where id="+userId);
 }
```

- 得到所有用户名

```
String[] getAllUsername(){
  return (String[]) jdbcTemplate.query(
    "select name from user",new ResultSetExtractor(){
        public Object extractData(ResultSet rs) throws SQLException{
          List list=new ArrayList();
          while(rs.next()){
```

```
                list.add(rs.getString(1));
              }
             return list.toArray(new String[list.size()]);
            }
      });
    }
```

可以看到这个 api 提供的 jdbc 封装比较灵活并很通用，使得原本复杂冗余的 JDBC 代码变成直白而简洁，这便是善用设计模式的结果。(Spring 的 jdbc 实用类大量使用 Template、Builder、Command、Strategy 等设计模式)

Spring 对 SQLException 的封装

另外值得一提的是 Spring 对 SQLException 的封装。org.springframework.dao.DataAccessException 是 Spring 中的 RuntimeException，作为数据访问相关异常 exception 继承树的根。Spring 对 SQLException 和 HibernateException 等异常作了封装，统一成 DataAccessException。这样在 DAO 中可以只抛出这个异常(及其子类)，而不会抛出 SQLExcception/HibernateException 等具体 API 的异常。

runtime exception vs. checked exception

把它设计成 Runtime Exception 而不是常见的 Checked Exception,这也是经过仔细考虑的。对于 RuntimeException 及其子类，不要求代码中显式地 try/catch。如我们经常遇到的 NullPointerException 和 NumberFormatException 便是 RuntimeException。假设他们是 checked exception, 那么我们编写任何一行对象的操作都要显式地 try/catch——即使在我们很明确不会出问题的情况下。如 `String s="a"; System.out.println(s.length());`如果 NullPointerException 是 checked exception, 则我们就不得不写成：

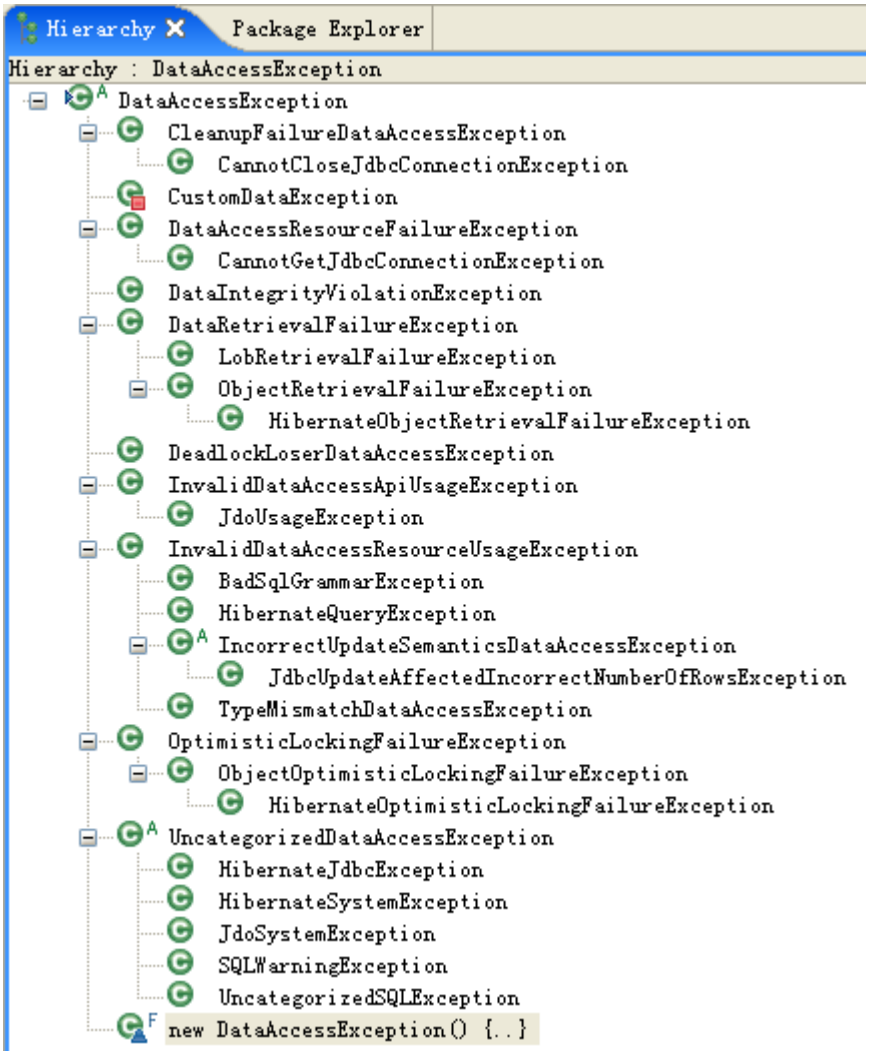
```
String s="a";
try{
  System.out.println(s.length());
}catch(NullPointerException e){
   //never happen
}
```

这样是罗嗦而不合理的。

对于 RuntimeException 与 Checked Exception 的取舍一直是 Java 社区讨论的话题。我比较赞同类似 SrpingFramework 开发人员的观点——尽量把异常设计成 RuntimeException, 免去代码中无谓的 try/catch block, 因为很多时候我们都无法在异常发生后挽回什么。当然不能让一个程序因为一个小小的异常（如 NumberFormatException）而非正常退出——应该在某些地方统一进行异常的处理。这样的代码不会有安全的隐患也更简洁。当然 Checked Exception 在某些情形下也是正确的选择，主要是异常很容易发生，或者异常的内容或抛出异常本身可以认为是方法的一种返回结果。

Spring 通过 SQLException 的语义来决定包装成什么 DataAccessException 的子类。 以往我们用 JDBC 直接读取数据，遇到的异常都是 SQLException，无论是主键冲突，还是超出列宽，或是 SQL 语法错误。而使用 Spring 的 JDBC Support，我们面对的是更有意义的异常，如 TypeMismatchDataAccessException、

OptimisticLockingFailureException、InvalidDataAccessApiUsageException 等。下图为其完整的继承树。



如此丰富的继承树有助于我们进行集中的异常处理。从实现上来说，SpringFramework 采集了各个数据库的不同 SQL Error Code，在遇到 SQLException 时分析其 error code，包装成对应的特定异常类。另外在在执行不同方法时抛出的 SQLException 也揭示了异常的语义，也有助于 SpringFramework 更好的完成异常的分类和包装。

多数据源

数据库固然是项目的基石， 但很多时候我们需要先收集和整理数据。原始的数据可能来自另一个数据库，或是一些 Excel 表格或 txt 文件，或需要监听某一端口定时产生的数据……

对于一种情形：原始数据不需要或是不能保存在数据库，我们可以采用 DAO 模式来屏蔽这种差异。还是原先那个用户信息的例子，如果数据保存在 xml 或是 excel 或是 txt 文件中，我们编写的 DAO 实现类只要使用对应的 api 读取这些数据。如果数据比较少，可以一次性读到内存（如 DAO 实现类的 findUserByID 被调用时，直接从内存中检索数据）。如果数据量比较大，则可能需要每次去访问一次数据源。

当然这是比较少见的情况，更多时候我们还是需要设计数据采集模块，将原始数据导入数据库，如有必

要还需要把更新过的数据再写回原始数据源。虽然这项工作与具体的业务需求比较紧密，没有特别通用的解决方法，但通过良好的设计还是可以达到一定程度的通用。
附录 A 中是我曾为一个实际项目的数据采集做的设计，仅作参考。

Web 层

Web 是 J2EE 项目最常见的表现方式，如何设计和开发出高质量的 Web 也是摆在开发人员面前的难题。

目前开发 Web 的一些困难是：

- Web 界面经常变化
  如经常需要变化一些页面布局、图片或样式表，但不一定改变工作流程。成功的应用可以接受这些修改而不需修改业务对象设置 Web 层的控制代码。
- Web 界面中涉及复杂的标签
  典型的动态网页中嵌有大量的表格，冗长的 JavaScript 以及用可视化工具生成的 HTML 代码（往往难以阅读）。一般只有相当少部分的内容是动态的。优秀的 Java 开发人员也没有必要掌握全面的页面制作和美工技能，这些工作应该由专业的页面制作和美工人员完成。
- Web 开发与普通 GUI 开发很不相同
  Web 开发所强调的是请求和响应，请求中只能带有 String 型的参数，页面上用户的输入往往无法控制，无法限制用户按一定顺序访问网页，而相反的是，传统的 GUI 提供丰富的控制和自定义控制的功能。
- Web 的测试比较困难

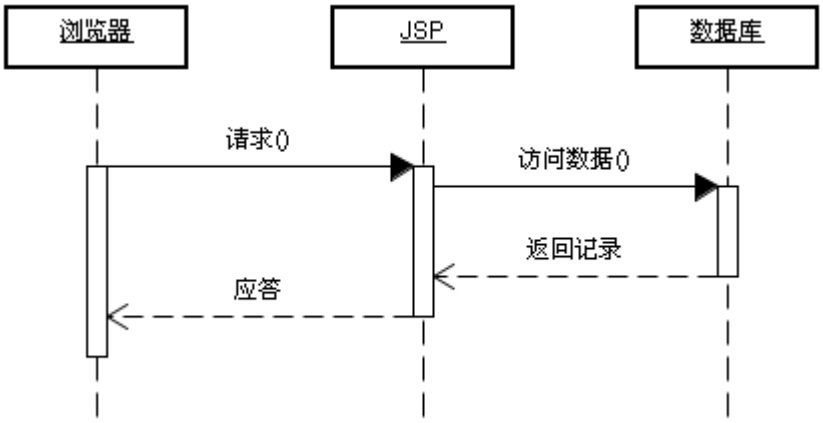
Web 开发也是一个很大的话题，无法在这么短的篇幅内讨论很多，限于个人经验，也无法开展很有深度和广度的探讨。以我对一些原则和技术及框架的理解，一个良好的 Web 开发模式应该能满足以下几点：

- 页面内容和显示分离
  一组数据，原先显示成 html table,现在要求显示成柱状图或饼状图，这种需求变更应该只需修改前台很少的代码。页面技术现在有很多，常见的 JSP 是一种，Velocity 应用也比较广泛，类似的还有 FreeMarker、Tiles、XML/XLST、XMLC 等。选择何种显示技术，对后台的影响应尽可能的小。另外 Web 框架应该能比较好地支持 PDF、Excel、Word 等视图的生成。
- Web 层的控制代码不应处理业务逻辑
  Web 只是系统的表现层，核心的处理逻辑应该封装在 Bean 或者 EJB 中，而 Web 只是接收和处理请求，把合适的请求转为向核心业务模块的调用。如果较好地完成这一点，Web 层便比较轻量和细薄，业务逻辑的内部实现变化并不会更改 Web 层的控制代码和显示。甚至从一个 B/S 架构转为 C/S 也是可行的。
- 对多语言的支持，对上传的支持等

MVC 的 Web 开发方式

MVC 是开发 Web 一种主流的模式，它良好地分离了视图、控制和数据。

朴素的 JSP 开发的模式可能是这样的



使用 JavaBean 甚至标签库以及 Servlet Filter 和 Servlet Listener 可以一定程度上把逻辑从 jsp 中分离出来，但只解决了少量的问题，我们还得处理流程控制、验证、更新应用程序状态等事情。

MVC 通过把问题分为三个类别来帮助解决单一模块所遇到的某些问题

- Model(模型)

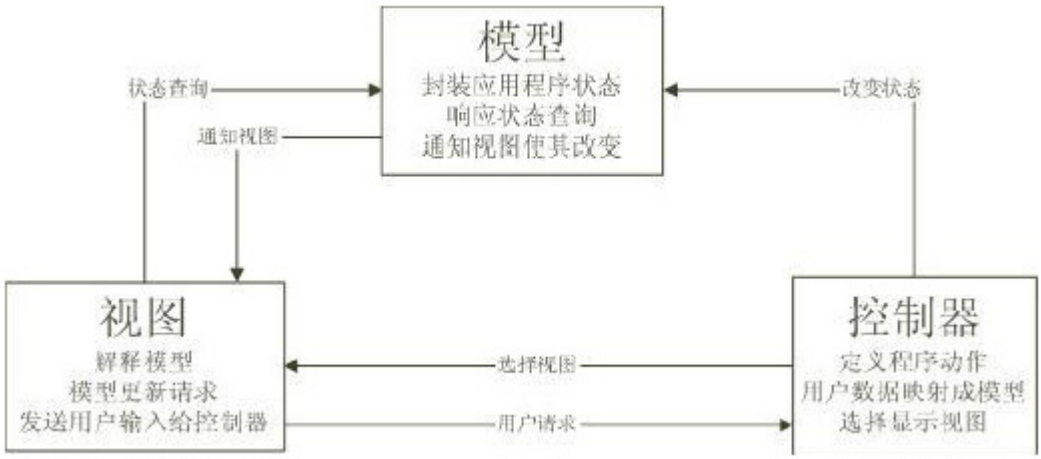
包含应用程序的核心功能。模型封装了应用程序的状态，它对视图或控制器一无所知。模型表示业务数据，或者业务逻辑

- View（视图）

提供模型的标识。它是应用程序的外观，使用户看到并与之交互的界面，视图可以访问模型的读方法，但不能访问写方法。此外，他对控制器一无所知。当更新模型时，通知和修改视图

- Controller(控制器）

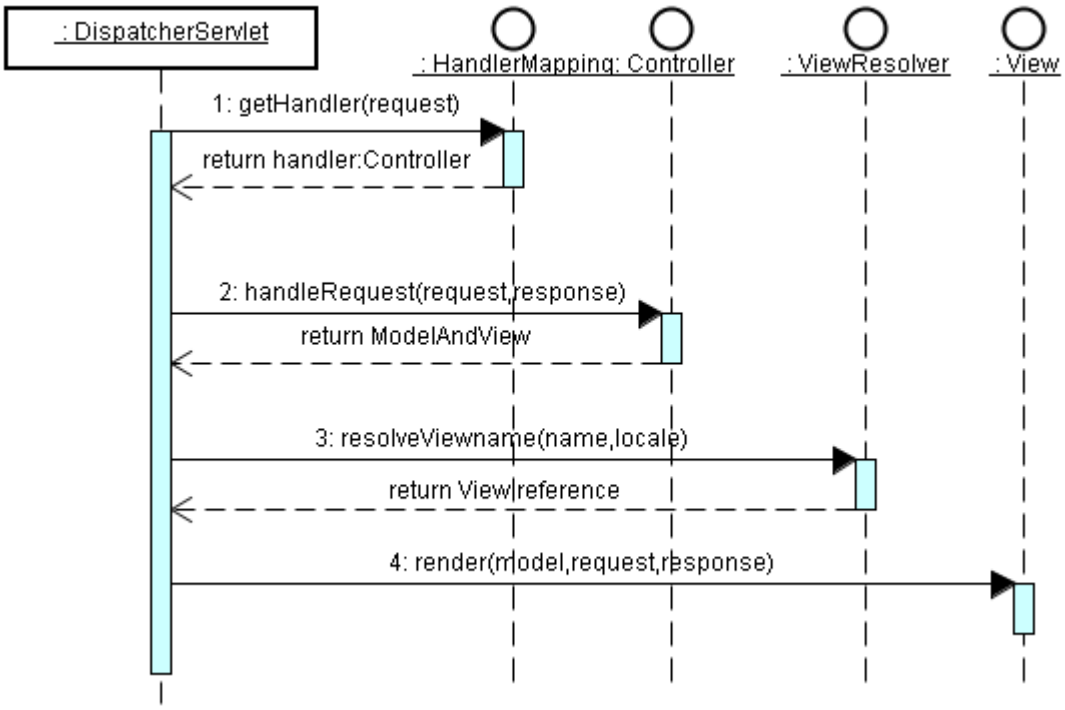
控制器对用户的请求/输入作反应，它创建并设置模型，选择要显示的视图



设计一个良好的 MVC Web 框架是件艰巨的任务，好在目前已经有不少现成的技术，如 Struts、WebWork、Maverick。其中 Struts 可能是出现时间最早使用也最多的 MVC 框架。不过我在这里介绍的是 SpringFramework 中的 WebMVC。因为通过比较，我觉得 Spring 在某些方面要比 Struts 更灵活和强大一些，对 Servlet API 甚至框架本身的依赖性都比较小。

SpringFramework WebMVC

SpringFramework WebMVC 的基本控制流如下：

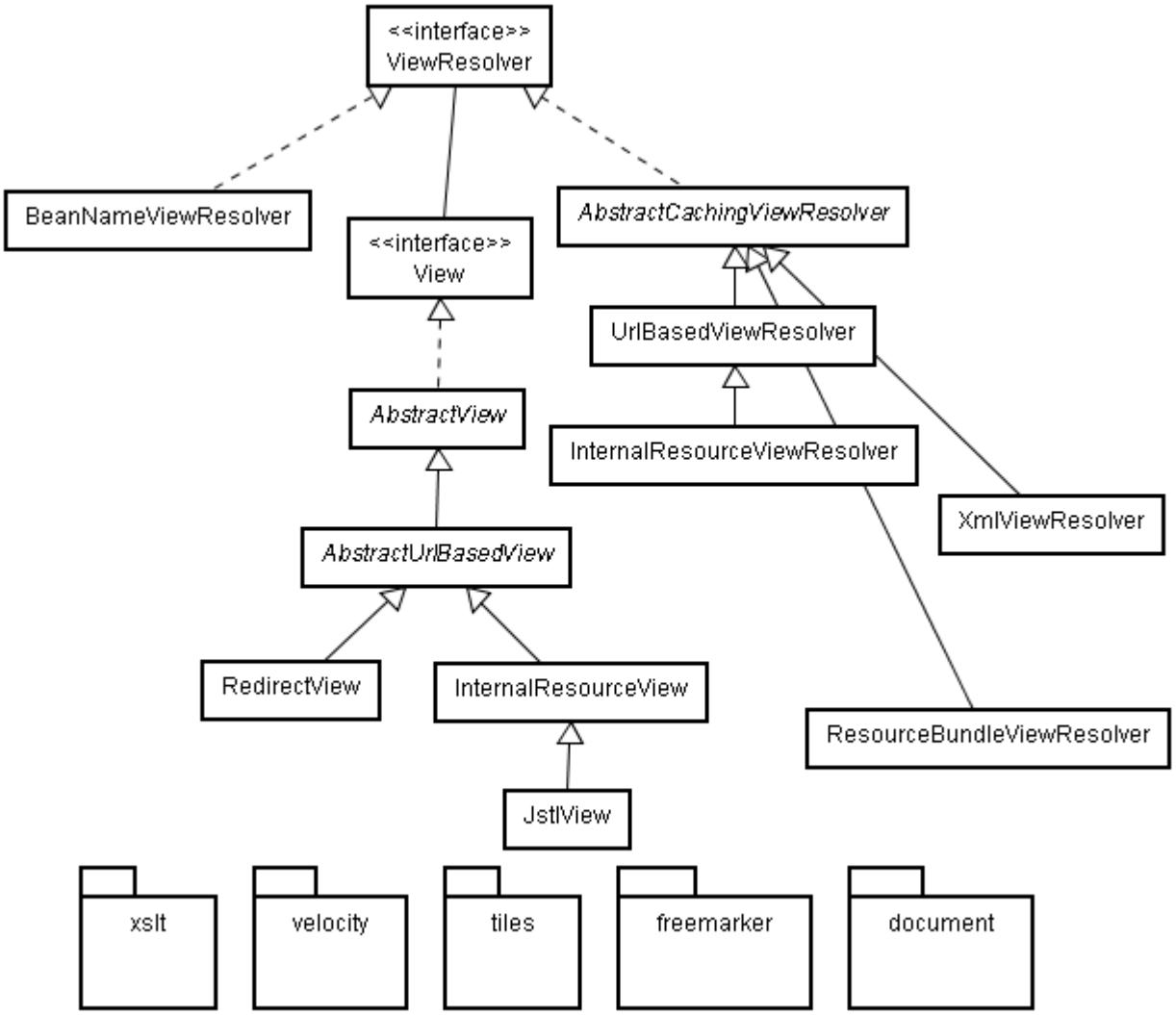


说明：

- 在接受一个 HTTP 请求后，根据 web.xml 中的配置，请求会交给一个 org.springframework.web.servlet.DispatcherServlet, 这是一个通用的 Servlet，并不处理实际的工作。正如名字所暗示的，它把请求 dispatch(转交)给其他对象，并返回最后的结果。这个类控制了 MVC 的主要流程。
- 和其他 MVC 框架一样，一个 Spring WebMVC 的应用需要一个或多个配置文件。DispatcherServlet 会调用当前的 HandlerMapping, 获知当前的请求应该由哪个 Controller 处理。由上面的 UML 图可以知道，除了 DispatcherServlet，其他的几个主要对象都是接口而非具体类。SpringFramework 大量使用面向接口的设计，这一点我将在下文介绍 Spring 的体系结构时谈到。
- 在获知应该由哪个 Controller 处理请求后，便调用该 Controller 的 handleRequest 方法，这也是 Controller 接口唯一必须实现的方法。而 Spring 提供了很多内建的 Controller 子类，可以更好的完成任务和简化开发，如常用的 MultiActionController 可以让多个 url 执行该 controller 的不同方法。
- handleRequest 的返回结果是一个 ModelAndView 对象，同时包括了 JavaBean 风格的 Model 和一个 View 的信息。Spring 中 Model 和 View 分的很清楚，这里出现的 ModelAndView 类只是限于 Java 语法，便于一个方法返回两个对象。
- Model 是一些普通的 Java 对象，每个对象有个字符串作为标示。一般来说，这些 Model 会使用 request.setAttribute(key,value)来绑定到 request 中，供 view 来使用。但是 Spring 的设计者充分考虑了别的可能性，比如我们如果选用 Velocity 或是 XMLC 作为前台的显示技术，它们未必使用 request.getAttribute 来获取数据。如 Velocity。在 Spring 得到 ModelAndView 后，会在 VelocityView 这个类中，把 Model 放入 context 中，供 VelocityEngine 来整合数据。这一点上，Spring 就要比 Struts 灵活些，Struts 由于仅关注 JSP，所以在 Struts 的 Controller 中需要手动把 Model 一个个 set 到 request 中，不但增加了程序员的负担，也为日后从 JSP 转成别的显示技术留下了隐患。
- 从ModelAndView中分离Model和View后，Spring需要把View转成一个具体的显示。在ModelAndView中，View往往是一个字符串或是某个子类加上一个名字，如new RedirectView(“a.htm”)。Spring调用ViewReslover来把这样的标识转为View对象。默认的ViewReslover是InternalResourceViewResolver，也就是以webapp中的相对路径作为view的名字。view也可以有很多别的reslover，如

ResourceBundleViewResolver会根据当前的locale选择一个view。这样，同一个显示界面，让美国人和中国人看到不同的页面布局是可能的。由于使用用一份数据（Model），所以这一层控制放在view的选择器也就是ViewReslover中是合理的。

- 最后，每个 View 都有一个 render 方法，根据 Model，以及 HTTP request 和 response 对象来生成最后的输出。Spring 内置很多 View 的子类，如 JstlView、AbstractExcelView、AbstractPdfView、AbstractXsltView 等。



除了这个基本的 MVC 框架，SpringWeb 还提供很多其他的功能，如 JavaBean 的绑定。也就是用户通过表单输入的内容，可以由 Spring 来自动 bind 成对应的 Java 对象，提交给 Controller，而避免以往复杂表单处理——通过 request.getParameter 把一个个参数取出再处理。不同于 Struts 中的 ActionForm，Spring 支持的这种表单对象是 JavaBean，因而业务系统中的 domain object 可以直接在 Web 中使用。Spring 也提供独立于 Web 也独立于 Spring Framework 的验证器，供 Controller 处理表单前验证有效性。
另外一个有别于 Struts 的是，Spring 没有提供 Struts 那么多自定义标签用于表单组件，而是使用 HTML 中的固有标签如<input type="text">来作为输入，在它外面套上 Spring 的 bind 标签，用于把该输入与 JavaBean 的某一个字段相绑定。第二部分中会有更深入的介绍。

此外，SpringWeb 对多语言多主题的支持也比较好。用户可以在浏览页面时随时更改界面语言和显示风格。另外 Spring 支持对页面的 cache 处理，通过简单的设置，可以在 Framework 中透明的完成对 view 的 cache。

另外，Spring 对上传的支持也比较新颖。目前有不少用于 servlet 的上传 api,如 JakartaCommons FileUpload,O'Reilly 的 cos upload 和 JSP SmartUpload 等。 在 Spring 中，可以在其特定的配置文件中声明使用何种上传组件，在实际处理含有 input type="file"的表单时，直接可以把 HttpRequest 强制转换为 MultipartHttpServletRequest，使用其 getFile 方法得到上传数据，而不用以往复杂的流处理。

更详细的 Spring Web 的介绍可参见相关网站和资料，本文的第二部分使用了 Spring 和其他技术完成了一个小型的网站，通过阅读其源码，可以对 Spring 有个更直观的认识。

SpringFramework 的简单介绍

在前面不少的模块中，我都提到了 SpringFramework，这是一个比较新的开源框架。在我第一次看到它后便被它深深吸引，因为很少有一个项目可以像它那样为 Java/J2EE 的开发提供强有力的基础架构，同时又提供丰富的实用类和扩展。Java 目前有很多第三方 api 和 Framework，但它们往往专注于 Java/J2EE 项目很小的一部分，如 Struts 基本上只是一个 Web 应用框架，Hibernate 是一个持久层技术。而 SpringFramework 被称作 J2EE 的软总线(soft bus), 不但在一些细节上为我们提供帮助，更在全局上令 J2EE 项目具有很高的灵活度。

Goal of SpringFramework

在具体介绍 Spring 之前，先提一下 SpringFramework 希望解决的问题

- 让 J2EE 开发变的简单、易用
- 业务应用不依靠 spring API
- 集成已存在的成熟应用解决方案
- 利用面向对象的优秀技术
- 促进良好的编程习惯
- 让测试业务应用变得简单、快速
- 有效地组织你的中间层对象
- 消除各式各样的配置文件
- 将面对接口编程做到实处
- 自由、不依赖框架
- 统一的数据存储方式
- 只选择你需要的

这里着重指出几个观点，我认为它们是 Spring 成功的主要原因，也是大型 J2EE 应用项目成功的要点。

利用面向对象的优秀技术

面向对象技术虽然不是解决一切问题的神兵利器，但确实可以令我们的解决方案得到比较大的益处。面向对象所追求的松散耦合，对象封装正是我们面对复杂需求的应对之策。这一点课堂和不少书籍上都有表述，但在实际项目的历练中才会愈发明白面向对象带来的好处。一定程度上，一个好的面向对象设计带来好的应用解决方案，往往也包括好的性能。

然而有的时候我们并不是从面向对象出发来进行设计和开发的，一个典型的情形就是 EJB 等复杂 J2EE

技术的使用。往往我们会陷入这样的误区——从一个 J2EE 的视角来分析和解决问题：面对一些用例，首先去考虑应该用哪些 Bean(EJB)来处理。 但其实 EJB 中一些特性是和面向对象相悖的——它是比较大粒度的组件，组件和组件间缺乏一般对象间丰富的关系，如继承和引用。而且 EJB 的其他限制（如不能新建线程、获取本地资源等）都会影响我们正常地面向对象设计。

我比较赞同的做法是，面对一个应用情形，首先使用传统的面向对象方法进行分析和设计，如果有必要把部分功能以特定的技术暴露出来，如 EJB。

EJB 等 J2EE 中重量级的技术目前正接受挑战。这里就不再赘述，有兴趣的可以在网上找到相关的讨论。

将面对接口编程做到实处

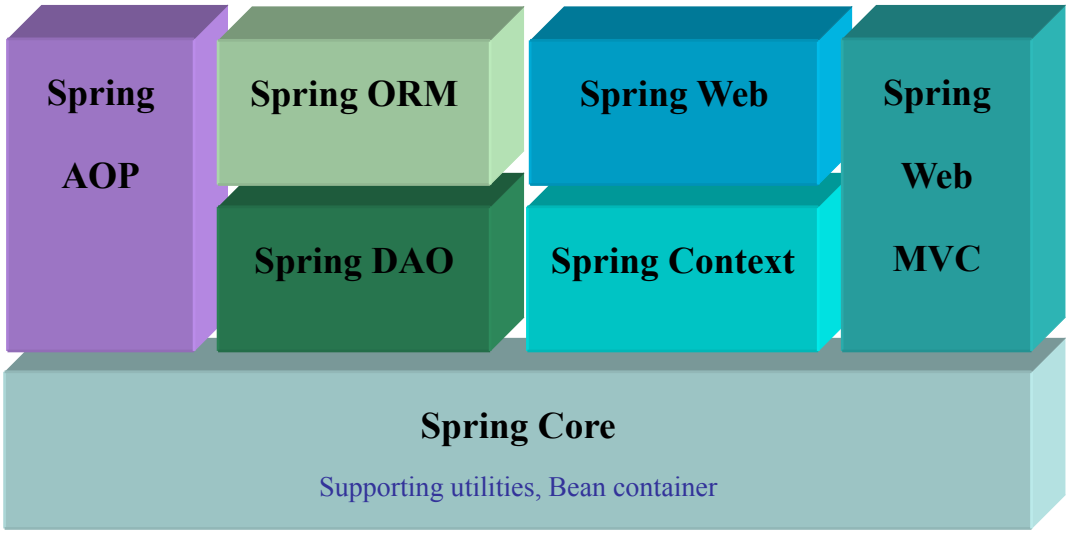
正像前面我们讨论多数据库和多数据源时使用的 DAO 模式一样，面向接口编程(而不是具体类)可以使我们的设计更为灵活，隐藏底层实现。Spring 中非常鼓励我们面向接口编程，我们保留一个对接口的引用而非具体类,这样我们可以很方便的从一种实现转为另一种实现。另外 Spring 很好的支持 AOP（Aspect Oriented Programming）, 如果面向接口编程，会意外的获得 AOP 带来的很多好处。(在第二部分的演示代码中，我们可以看到基于 AOP 的事务管理和权限控制)。Struts 一定程度上没有很好地做到面向接口编程，而是用了很多具体继承，这使得基于 Struts 的应用甚至 Struts 本身欠缺灵活性，也不利于测试。

将单元测试变的简单

JUnit 为代表的单元测试工具可以大大简化我们的单元测试工作。无论是使用极限编程所倡导的测试驱动开发（TDD：Test-Driven Development）还是传统的(先设计，再编码，然后测试以次迭代)，单元测试的使用会大大提高我们开发的质量（测试驱动开发甚至可以改善我们的设计，令暴露的接口更加合理）。
一个复杂的 Java/J2EE 项目，尤其是核心的业务处理模块，良好的单元测试会令每个细节都比较健壮，这对提高整体的质量降低风险是有益的。而前面提到的良好的面向对象设计和面向接口编程会令单元测试变得更简单，加上 Spring 提供的 BeanFacotry, IoC Container, 会令资源的分配更为灵活。即使不启动 Web Server，我们也可以充分测试 Controller，甚至借助 Mock 技术测试与数据库、J2EE Container 相关的模块。

SpringFramework 的概貌

下面是 SpringFramework 的概貌:



Spring Core	主要提供 Bean Factory 和 IoC Container，以及若干实用类
Spring AOP	提供运行时的 AOP 支持和源代码级的元数据(metadata)支持
Spring DAO	事务支持，JDBC 包装类，DAO 模式的支持
Spring ORM	多种常见 ORM 技术的支持，如 JDO、Hibernate、iBatis
Spring Contex	对 JNDI，EJB，JavaMail，验证和 UI
Spring Web	基于 Web 的 Context，透明的组件上传，Web 开发实用类
Spring Web MVC	MVC 框架，JSP/Excel/Pdf/Xml/Velocity 等多种视图技术的支持

从上面这张图上可以看到，SpringFramework 的模块性很强，你可以选择使用它的一个或多个功能，如仅使用它的 BeanFactory/IocContainer，甚至只使用它对 JDBC 的封装。如果开发团队中比较熟悉 Struts，则可以继续使用 Struts 作为 Web 框架，而是用 Spring 提供的 JavaBean 管理和事务支持。 所以，虽然 Spring 提供了 One-Stop Shop 的服务，但使用多少完全是开发人员的选择，这一点上不像一些别的技术，要么不用，要么都用。

BeanFactory 和 IoC 容器

上文提到了 BeanFactory 和 IoC 容器，这是 Spring 中很重要的一个概念，是 Spring 的精髓。

什么是 BeanFactory?

经典的 Factory 设计模式分为工厂方法、简单工厂和抽象工厂。把创建什么对象怎么创建的职责从类本身或是客户程序员处转到专门的工厂类中，确实提供了更好的封装和重用。JDK 中也随处可见这样的例子，如 SwingAPI 中的 BorderFactory。但这一设计模式的问题在于工厂的责任过于重大——它需要了解很多被生产类的信息，这甚至一定程度上违背了面向对象一个很基本的原则——开闭原则（Software entities should be open for extension,but closed for modification.）。一旦类发生了改变或是增加了新的类，往往工厂也得修改。为此有人主张借助 Java 强大的反射功能来以不变应万变，动态生成类的实例。
比如原本的简单工厂为

```
public IStuff factory(String name)..
   if("A".equals(name)) return ...
```

```
   if("B".equals(name)) return ..
   ..
}
```

使用反射，可以简化为

```
public IStuff factory(String className)..
  Class clazz=Class.forName(className);
  Object obj=clazz.newInstance();
  if(obj instanceof IStuff) return obj;
  ..
}
```

这样工厂就只承担简单的类型判断等责任，足以应付将来的各种变化。

Spring 正是把这个简单而有效的想法赋予更灵活的实现。 有一个 BeanFactory 读取配置文件(一般是 XML 文件)按要求新建 Bean 的实例，并加以维护。

```
<beans>
<bean id="exampleBean" class="eg.ExampleBean"/>
<bean id="anotherExample" class="eg.ExampleBeanTwo"/>
</beans>
```

上面的例子中新建了两个对象，分别是 eg.ExampleBean 和 eg.ExampleBeanTwo 的两个实例。在应用程序中使用下面的代码来访问这两个对象:

```
InputStream input = new FileInputStream("beans.xml");
BeanFactory factory = new XmlBeanFactory(input);
ExampleBean eb =(ExampleBean)factory.getBean("exampleBean");
```

默认的，新建的 Bean 是作为 Singleton 的，也就是说系统中所有出现 `factory.getBean("exampleBean")` 的地方都返回同一个实例。这也是 Spring 所具有的一个特点——避免过度的 Singleton 模式的使用。

Singleton 作为一个简单而有效的模式，确实被我们经常用到，但他的问题在于：

- Singleton 是具体类，欠缺接口的很多优势。这一点参见前面提到的面向接口编程的好处。
- Singleton 的使用造成难以与接口打交道，如一个核型的业务是一个接口，提供多种实现，而在 Singleton 中往往需要给出具体的实现，增加了硬编码依赖性
- 每个 Singleton 需要自己查找它的配置参数，通过属性、JNDI、JDBC 等。 系统如果出现很多 Singleton，会令配置管理不一致，并每个 Singleton 中保留了与业务逻辑不相关的大量关于配置的代码

上述问题，使用 BeanFactory 可以得到较好的解决。因为

- Spring 提供统一而强大的语法，在配置文件中统一进行 Singleton 的建立
- 可以新建一个 Bean，它是实现某个接口的具体类的实例，而把它赋给另一个 Bean，后者的方法中只要求一个接口的对象，因而不关心前者具体的类型。这样，在必要时我们只需要改变前者的类名，不需要改变 Bean 的 ID，也不需要改变后者的配置和代码，便完成了内部实现方法的改变。这也是面向接口编程在 Spring 的体现。
- 在运行时新建类，如有必要可以在第一次访问 Singleton 时新建实例，得到少量的性能收益。

除了 Singleton 模式的新建，BeanFactory 也支持 prototype，也就是按配置为模版，每次访问 getBean 时访问一个相同内容的对象，这在有时候也是有必要的。

需要指出的，Spring 的基础是 BeanFactory，但是上层的应用可以对这个细节不关心，也就是我们在使用 Spring 做应用时，可以不去调用 getBean 这些底层的方法。比如在第二部分的 Demo 中，我们的 Web 应用就是使用 Spring Web MVC,没有直接接触 BeanFactory。

什么是 IoC?

IoC，Inversion of Control 现在被正式更名为 Dependency Injection。这其实也是一个很简单的模式，具体例子便会了解到他的简单和实用。
比如我们需要在一个类中调用一个 EJB 的一个方法。一种做法是在这个类中(类名比如是 Foo)，创建 context，给出连接应用服务器的一些参数，然后 lookup home interface, 再 create 一个远程对象，调用远程方法。而使用 IoC 模式的 Foo 类格外简单

```
public class Foo..
  private RemoteInterfaceOfEJB bean;
  public void setBean(RemoteInterfaceOfEJB ejb){this.bean=ejb;}
  public void foobar()..
    bean.invokeSomeOp();
  }
}
```

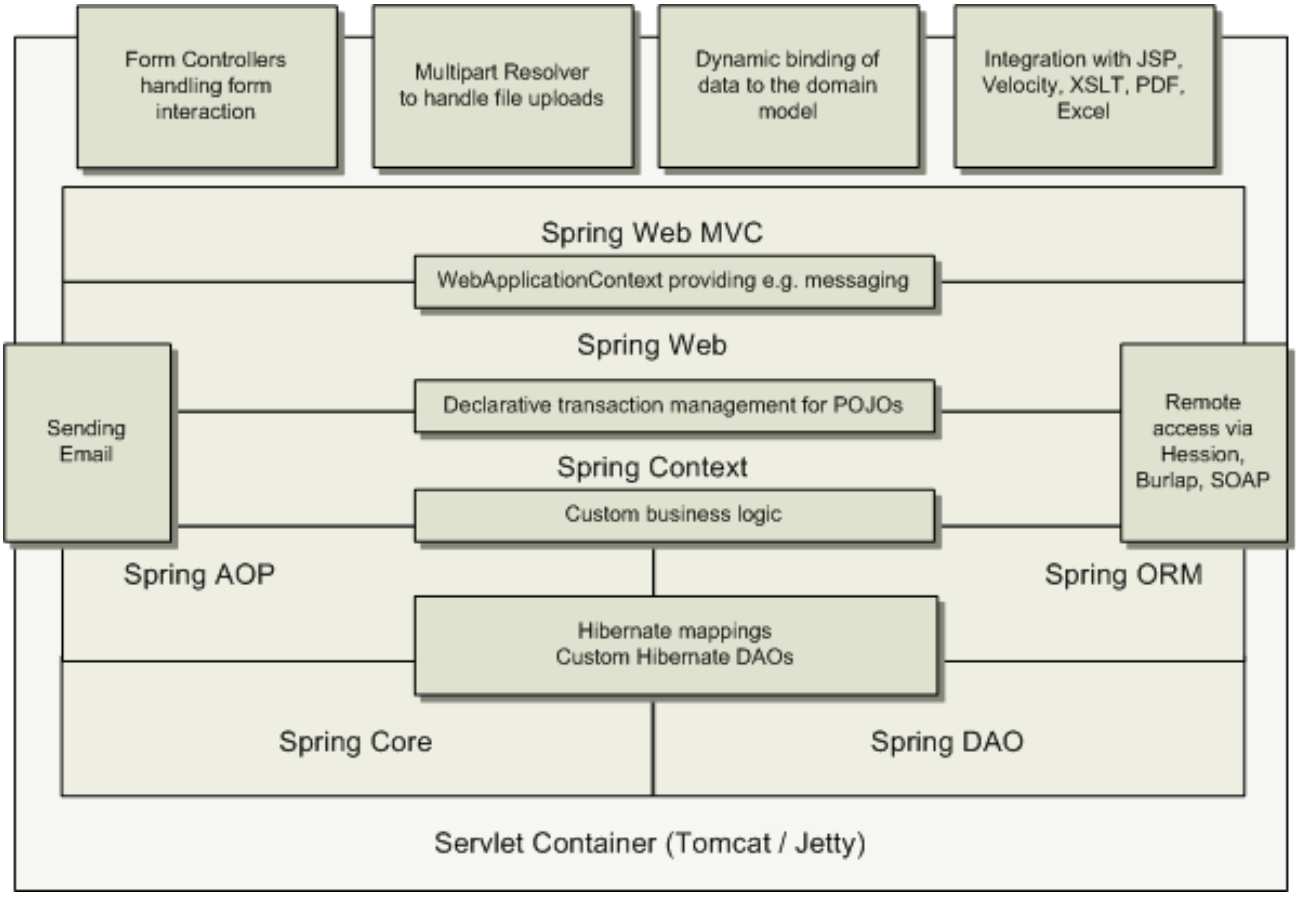
可以看到新的 Foo 中省去了大量查找资源或是释放资源的代码，而是依赖外界给它提供 EJB 的远程接口直接调用。这样 Foo 中的责任更加清晰，对我们开发和维护都有好处。那么这个 EJB 应该由谁创建和维护呢，答案是 IoC Container。SpringFramework 就提供了这样一个 IoC Container, 用户接口就是前面提到的 BeanFactory。 对于 IoC 的各种变种和深层次讨论，限于篇幅不在此讨论，有兴趣的可以查看相关书籍和网上的讨论。

除去 BeanFactory、IoC，Spring 还有很多重要的部分，就像前面的整体架构图中看到那样。不过限于篇幅，就不再一一介绍了。SpringFramework 本身已经提供很多框架性的支持和不少的实用类，它也在积极地改进和发展中，并有不少项目支持和扩展 Spring，如基于 Spring 的 Rich Client Platform 和权限框架，Spring 的 IDE 插件，Microsoft .Net 平台的 SpringFramework 移植等。在第二部分中，我们会演示如何使用 Spring 和其他优秀的技术完成一个实际的项目。

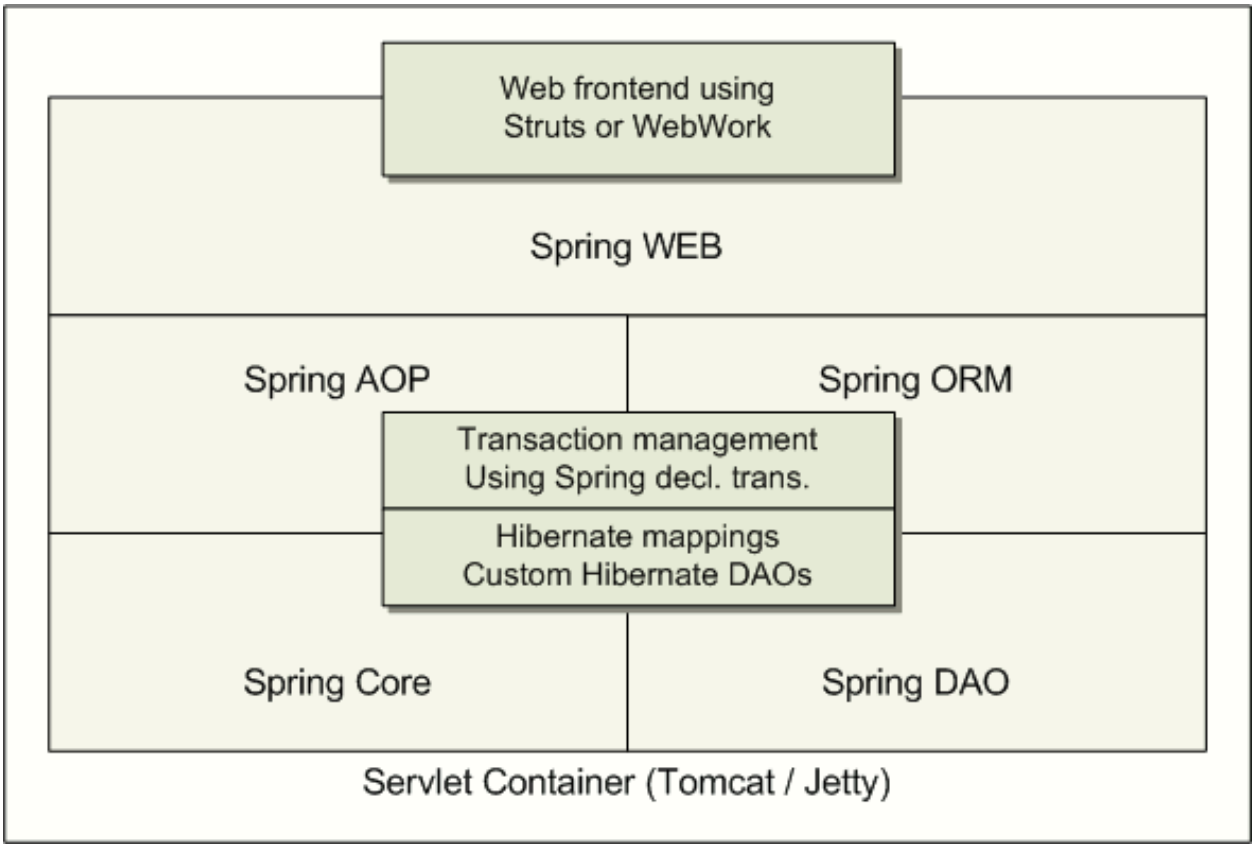
最后，给出基于 Spring 的若干技术构架

Spring 技术构架

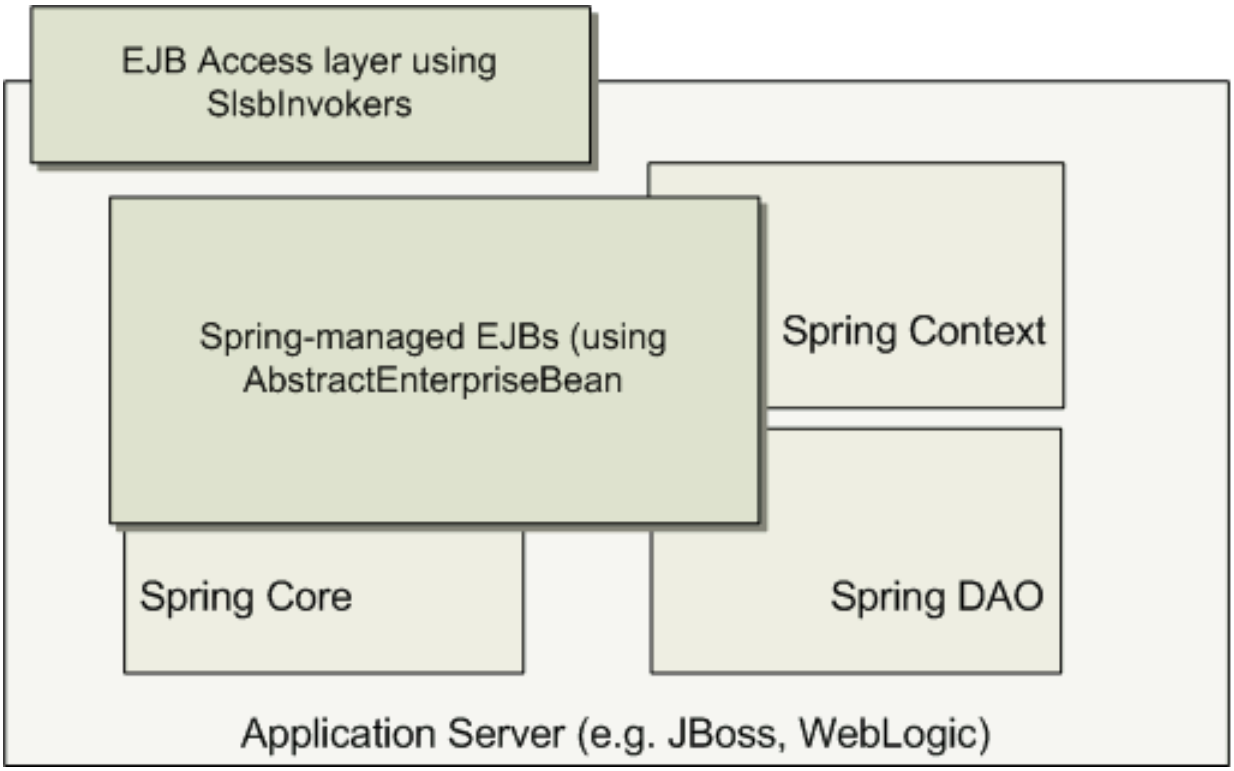
Power the whole application with Spring



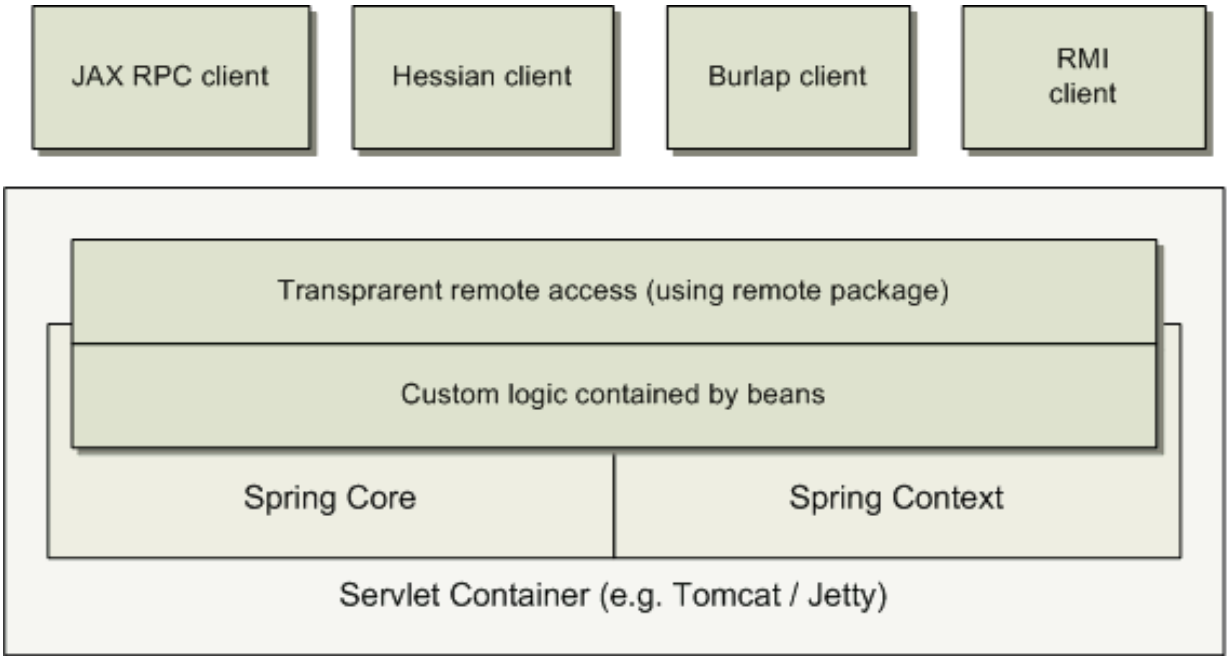
Integrate with third-party web application frameworks



Use Spring to manage EJBs



Remoting through SOAP, RMI, Web services



参考资料

- Expert One-on-One: J2EE Design and Development by Rod Johnson（Wrox, 2002.10）
- http://www.hibernate.org
- http://www.springframework.org
- http://jakarta.apache.org
- http://martinfowler.com/articles/injection.html

第二部分 **Demo** 系统

附录 **A**