

```

    }
}
}
```

现在，我们可以实现如下所示的聚合器 Restaurant。在这里，只使用了基本的属性。可以根据需要添加任意多的属性，还可以添加你想要的其他功能：

```

public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();
    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }
}
```

存储库的实现

现在，我们可以来实现在这一章中了解的存储库模式。开始时，你将首先创建两个接口 Repository 和 ReadOnlyRepository。ReadOnlyRepository 将被用于提供只读操作的抽象，而 Repository 抽象将用于执行所有类型的操作：

```

public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();

}
```

基于该接口，我们可以创建会做额外的操作，如添加、删除和更新的存储库抽象：

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {  
  
    void add(TE entity);  
  
    void remove(T id);  
  
    void update(TE entity);  
}
```

前面定义的存储库抽象可以用适合的方式来持久化你的对象。在持久性代码，即基础设施层的一部分所做的改变不会影响领域层代码，因为合同和抽象是由领域层定义的。领域层使用抽象类和接口，消除对直接具体的类的使用，并提供松耦合。出于演示目的，我们可以简单地使用仍然保留在内存的映射来持久化对象：

```
public interface RestaurantRepository<Restaurant, String> extends  
Repository<Restaurant, String> {  
  
    boolean ContainsName(String name);  
}  
  
public class InMemRestaurantRepository implements RestaurantRepository  
<Restaurant, String> {  
  
    private Map<String, Restaurant> entities;  
  
    public InMemRestaurantRepository() {  
        entities = new HashMap();  
    }  
  
    @Override  
    public boolean ContainsName(String name) {  
        return entities.containsKey(name);  
    }  
  
    @Override
```

```
public void add(Restaurant entity) {
    entities.put(entity.getName(), entity);
}

@Override
public void remove(String id) {
    if (entities.containsKey(id)) {
        entities.remove(id);
    }
}

@Override
public void update(Restaurant entity) {
    if (entities.containsKey(entity.getName())) {
        entities.put(entity.getName(), entity);
    }
}

@Override
public boolean contains(String id) {
    throw new UnsupportedOperationException("Not supported yet.");
    //为了修改生成的方法的正文，选择 Tools | Templates。
}

@Override
public Entity get(String id) {
    throw new UnsupportedOperationException("Not supported yet.");
    //为了修改生成的方法的正文，选择Tools | Templates。
}

@Override
public Collection<Restaurant> getAll() {
    return entities.values();
}

}
```

服务的实现

用前述的方法，可以将领域服务的抽象分割成两个部分：主服务抽象和只读服务的抽象：

```
public abstract class ReadOnlyBaseService<TE, T> {

    private Repository<TE, T> repository;

    ReadOnlyBaseService(Repository<TE, T> repository) {
        this.repository = repository;
    }

    ...
}
```

现在，我们可以使用这个 `ReadOnlyBaseService` 来创建 `BaseService`。在这里，我们使用依赖注入模式，通过一个构造函数来映射具有抽象的具体对象：

```
public abstract class BaseService<TE, T> extends
ReadOnlyBaseService<TE, T> {

    private Repository<TE, T> _repository;

    BaseService(Repository<TE, T> repository) {
        super(repository);
        _repository = repository;
    }

    public void add(TE entity) throws Exception {
        _repository.add(entity);
    }

    public Collection<TE> getAll() {
        return _repository.getAll();
    }
}
```

现在，在定义了服务抽象服务后，我们就可以用下列方式实现 `RestaurantService`：

```
public class RestaurantService extends BaseService<Restaurant, BigInteger> {  
  
    private RestaurantRepository<Restaurant, String> restaurantRepository;  
  
    public RestaurantService(RestaurantRepository repository) {  
        super(repository);  
        restaurantRepository = repository;  
    }  
  
    public void add(Restaurant restaurant) throws Exception {  
        if (restaurantRepository.ContainsName(restaurant.getName())) {  
            throw new Exception(String.format("There is already a  
product with the name - %s", restaurant.getName()));  
        }  
  
        if (restaurant.getName() == null || "".equals(restaurant.getName())) {  
            throw new Exception("Restaurant name cannot be null or empty  
string.");  
        }  
        super.add(restaurant);  
    }  
}
```

同样，可以编写其他实体的实现。此代码是一个基本的实现，可以在生产代码中添加各种实现和行为。

小结

本章学习了 DDD 的基本原理，也探讨了多层架构和人们使用 DDD 开发软件可用的不同模式。这时，你可能会意识到领域模型设计对软件的成功是非常重要的。最后，还使用餐馆订座系统来演示了一个领域服务的实现。

在下一章，将学习如何将此设计用于实现示例项目。此示例项目的设计说明是从最后一章摘录的，并将把 DDD 用于生成微服务。这一章不仅包括编码，还包括微服务的不同方面，如构建、单元测试和包装。在下一章末尾，示例微服务项目将可用于部署和使用。

4

实现微服务

本章引导你从我们的示例项目——在线餐馆订位系统 (**OTRS**) 的设计阶段进入实现阶段。在这里，你将使用上一章所述的相同设计，并增强它，以建立微服务。在这一章的结尾，你不仅将学会实现此设计，还将学到微服务的不同方面——构建、测试和包装。虽然重点是建立和实现 Restaurant 微服务，你可以使用同样的方法来建立和实现 OTRS 中用到的其他微服务。

在这一章，我们将介绍以下主题：

- OTRS 概述
- 开发和实现微服务
- 测试

我们将使用上一章所述的领域驱动设计关键概念。在上一章，你看到了如何将领域驱动设计用于开发使用了核心 Java 的领域模型。现在，我们将从一个示例领域实现转向一个 Spring 框架驱动的实现。你将会利用 Spring Boot 来实现领域驱动设计概念并将它们从核心 Java 转换为基于 Spring 框架的模型。

此外，我们还会使用 Spring Cloud，它提供了一个云就绪的解决方案。Spring Cloud 也使用 Spring Boot，它允许你使用嵌入式应用程序容器，这依靠你的服务内的 Tomcat 或 Jetty，被打包为一个 JAR 文件或 WAR 文件。这个 JAR 作为一个单独的进程执行，也就是将对所有请求提供服务和响应，并指向此服务中定义的端点的一个微服务。

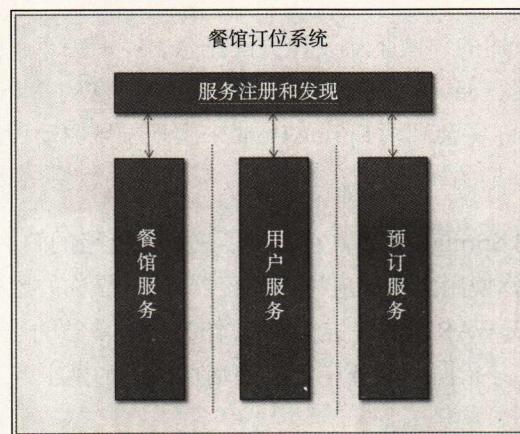
Spring Cloud 也可以方便地与 Netflix Eureka 集成，这是一种服务注册和发现组件。OTRS 将它用于注册和发现微服务。

OTRS 概述

基于微服务的原则，我们需要为每个可以独立执行的功能都建立单独的微服务。观察 OTRS 之后，我们可以轻松地把 OTRS 分为三个主要的微服务——餐馆服务、预订服务和用户服务，还可以在 OTRS 中定义其他的微服务。我们的重点是这三个微服务，要点是让它们独立，包括拥有自己单独的数据库。

我们可以总结出这些服务的功能，如下所示：

- **餐馆服务 (Restaurant service)**：此服务提供餐馆资源的如下功能——创建、读取、更新、删除 (CRUD) 操作和基于标准的搜索。它提供餐馆和餐桌之间的关联，餐馆还将提供对 Table 实体的访问。
- **用户服务 (User service)**：顾名思义，此服务允许最终用户在 User 实体上执行 CRUD 操作。
- **预订服务 (Booking service)**：此服务使用的餐馆服务和用户服务对预订执行 CRUD 操作。它会使用餐馆搜索，与其相关联的餐桌查找，并基于餐桌在指定的时段的可用性对其进行分配。它会建立 Restaurant/Table 和 User 之间的关系。



不同的微服务、注册和发现

上图显示每个微服务是如何独立地工作的。这是微服务能够分别被开发、增强、维护，而不影响其他服务的原因。这些服务可以分别有自己的分层架构和数据库，并不限于使用相同的技术、框架和语言来开发这些服务。你也可以在任何给定时间点上引入新的微服务。例如，为会计目的，我们可以引入向餐馆服务公开记账的会计服务。同样，分析和报告也都是可以集成和公开的其他服务。

出于演示目的，我们将只实现前面的关系图中所示的三个服务。

开发和实现微服务

我们将使用上一章所述的领域驱动实现和方法来实现利用 Spring Cloud 的微服务。让我们重温以下关键的工件：

- **实体：**这些都是可识别并保持产品或服务的状态不变的对象的类别。这些对象不(**NOT**)由它们的属性定义，而由它们的身份和线程的连续性定义。
实体拥有诸如身份、连续性的线程，以及不能确定其身份的属性等特征。**值对象(VO)**只有属性且没有概念上的身份。最佳的做法是把值对象保持为不可变的对象。在 Spring 框架中，实体是纯 Pojo，因此我们也把它们作为 VO 来使用。
- **服务：**这些都是技术框架中常见的，也用于领域驱动设计的领域层。服务对象不具有内部的状态，它的唯一目的是提供针对领域的行为。服务对象提供不能与特定实体或值对象相关联的行为。服务对象可能会给一个或多个实体或值对象提供一个或多个相关行为。在领域模型中显式定义服务是最佳做法。
- **存储库对象：**存储库对象是领域模型的一部分，它与存储区，如数据库、外部来源等交互，以获取持久化的对象。当存储库收到对某个对象引用的请求时，它返回现有的对象引用。如果存储库中不存在所请求的对象，那么它从存储区获取此对象。

下载示例代码

本书序言中提到下载代码包的详细步骤，请查看。



本书的代码压缩包也驻留在 GitHub 上，位于

<https://github.com/PacktPublishing/Mastering-Microservices-with-Java>。在我们丰富的书籍和视频目录中也有其他的代码包，请查看 <https://github.com/PacktPublishing/>。

- 每个 OTRS 微服务 API 都表示一个 REST 式的 web 服务。OTRS API 使用 HTTP 动词，如 GET、POST 等，以及一个 REST 式的端点结构。请求和响应的有效载荷都被格式化为 JSON。如果需要，还可以使用 XML。

餐馆微服务

Restaurant 微服务将以 REST 端点的方式对外部世界公开提供使用。我们会在 Restaurant 微服务示例中找到下列端点，可以根据要求添加任意多的端点：

端点	GET /v1/restaurants/<Restaurant-Id>	
参数		
名称	说明	
Restaurant_Id	路径参数，表示与此 ID 相关联的唯一餐馆	
请求		
属性	类型	说明
无		
响应		
属性	类型	说明
Restaurant	Restaurant 对象	与给定的 ID 相关联的 Restaurant 对象

端点	GET /v1/restaurants/	
参数		
名称	说明	
无		
请求		
属性	类型	说明
Name	字符串	表示餐馆的名称或名称的子字符串的查询参数
响应		
属性	类型	说明
Restaurants	Restaurant 对象数组	返回其名称中包含给定名称值的所有餐馆

端点	POST /v1/restaurants/	
参数		
名称	说明	
无		
请求		
属性	类型	说明
Restaurant	Restaurant 对象	餐馆对象的 JSON 表示
响应		
属性	类型	说明
Restaurant	Restaurant 对象	一个新创建的 Restaurant 对象

同样，我们可以添加许多端点以及它们的实现。出于演示目的，我们将使用 Spring Cloud 来实现前面的端点。

控制器类

Restaurant 控制器使用 @RestController 注解来建立这个餐馆服务端点。我们已经在第 2 章通览了 @RestController 的详细信息。@RestController 是用于资源类的类级别注解，它是 @Controller 和 @ResponseBody 的组合，它返回领域对象。

API 版本控制

在我们进入下一主题前，我想介绍一下在 REST 端点上使用的 v1 前缀，它表示 API 版本。我也想强调一下 API 版本控制的重要性。因为随着时间的推移，API 会变化，所以版本控制 API 是重要的。随着时间的推移，你的知识和经验更丰富了，从而需要对你的 API 做出更改，API 的更改可能会破坏现有客户端的集成。

因此，有很多种管理 API 版本的办法。其中一种是在路径中使用版本，还有一种方法是使用 HTTP 标头。HTTP 标头可以是表示调用的 API 版本的自定义请求标头或 Accept 标头。请参阅 Packt 出版的 Bhakti Mehta 编写的《RESTful Java Patterns and Best Practices》一书，请访问 <https://www.packtpub.com/application-development/restful-javapatterns-and-best-practices>，以获得更多的信息。

```
@RestController
@RequestMapping("/v1/restaurants")
```

```
public class RestaurantController {  
  
    protected Logger logger = Logger.getLogger(RestaurantController.  
class.getName());  
  
    protected RestaurantService restaurantService;  
  
    @Autowired  
    public RestaurantController(RestaurantService restaurantService) {  
        this.restaurantService = restaurantService;  
    }  
  
    /**  
     * 获取具有指定名称的餐馆。支持不区分大小的部分  
     * 匹配。所以，<code>http://.../restaurants/rest</code> 将会找出  
     * 在其名字中有大写或小写 'rest' 的任何餐馆。  
     *  
     * @param name  
     * @return 一个不为空、非空的餐馆集合。  
     */  
    @RequestMapping(method = RequestMethod.GET)  
    public ResponseEntity<Collection<Restaurant>> findByName(@  
RequestParam("name") String name) {  
  
        logger.info(String.format("restaurant-service findByName() invoked:{}  
for {} ", restaurantService.getClass().getName(), name));  
        name = name.trim().toLowerCase();  
        Collection<Restaurant> restaurants;  
        try {  
            restaurants = restaurantService.findByName(name);  
        } catch (Exception ex) {  
            logger.log(Level.WARNING, "Exception raised findByName  
REST Call", ex);  
            return new ResponseEntity< Collection<  
Restaurant>>(HttpStatus.INTERNAL_SERVER_ERROR);  
        }  
        return restaurants.size() > 0 ? new ResponseEntity<
```

```
Collection< Restaurant>>(restaurants, HttpStatus.OK)
        : new ResponseEntity< Collection<
Restaurant>>(HttpStatus.NO_CONTENT);
}

/**
 * 获取具有指定ID的餐馆。
 * <code>http://.../v1/restaurants/{restaurant_id}</code> 将返回
 * 具有指定ID的餐馆。
 *
 * @param restaurant_id
 * @return 一个不为空、非空的餐馆集合。
 */
@RequestMapping(value = "/{restaurant_id}", method =
RequestMethod.GET)
public ResponseEntity<Entity> findById(@PathVariable("restaurant_
id") String id) {

    logger.info(String.format("restaurant-service findById()
invoked:{} for {} ", restaurantService.getClass().getName(), id));
    id = id.trim();
    Entity restaurant;
    try {
        restaurant = restaurantService.findById(id);
    } catch (Exception ex) {
        logger.log(Level.SEVERE, "Exception raised findById REST
Call", ex); // findById REST调用引发异常
        return new ResponseEntity<Entity>(HttpStatus.INTERNAL_
SERVER_ERROR);
    }
    return restaurant != null ? new ResponseEntity<Entity>(restaur
ant, HttpStatus.OK)
        : new ResponseEntity<Entity>(HttpStatus.NO_CONTENT);
}

/**
```

```
* 添加具有指定信息的餐馆。  
*  
* @param Restaurant  
* @返回非空的餐馆。  
* @如果根本没有匹配项，则抛出RestaurantNotFoundException。  
*/  
  
@RequestMapping(method = RequestMethod.POST)  
public ResponseEntity<Restaurant> add(@RequestBody RestaurantVO  
restaurantVO) {  
  
    logger.info(String.format("restaurant-service add() invoked:  
%s for %s", restaurantService.getClass().getName(), restaurantVO.  
getName()));  
  
    Restaurant restaurant = new Restaurant(null, null, null);  
    BeanUtils.copyProperties(restaurantVO, restaurant);  
    try {  
        restaurantService.add(restaurant);  
    } catch (Exception ex) {  
        logger.log(Level.WARNING, "Exception raised add Restaurant  
REST Call "+ ex); // add Restaurant REST调用引发异常  
        return new ResponseEntity<Restaurant>(HttpStatus.  
UNPROCESSABLE_ENTITY);  
    }  
    return new ResponseEntity<Restaurant>(HttpStatus.CREATED);  
}  
}
```

服务类

RestaurantController 使用 RestaurantService。RestaurantServiceis 定义了 CRUD 和一些界面搜索操作，定义如下：

```
public interface RestaurantService {  
  
    public void add(Restaurant restaurant) throws Exception;
```

```
public void update(Restaurant restaurant) throws Exception;

public void delete(String id) throws Exception;

public Entity findById(String restaurantId) throws Exception;

public Collection<Restaurant> findByName(String name) throws Exception;

public Collection<Restaurant> findByCriteria(Map<String,
ArrayList<String>> name) throws Exception;
}
```

现在，我们可以实现刚定义的 RestaurantService。它还扩展了在上一章中创建的 BaseService。我们使用@Service注解将其定义为一种服务：

```
@Service("restaurantService")
public class RestaurantServiceImpl extends BaseService<Restaurant,
String>
    implements RestaurantService {

    private RestaurantRepository<Restaurant, String>
restaurantRepository;

    @Autowired
    public RestaurantServiceImpl(RestaurantRepository<Restaurant,
String> restaurantRepository) {
        super(restaurantRepository);
        this.restaurantRepository = restaurantRepository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurant.getName() == null || "".equals(restaurant.
getName())) {
            throw new Exception("Restaurant name cannot be null or
empty string."); // 餐馆名称不能为 null 或空字符串。
        }
    }
}
```

```
if (restaurantRepository.containsName(restaurant.getName())) {
    throw new Exception(String.format("There is already a
product with the name - %s", restaurant.getName()));
} //已经有一种同名产品

super.add(restaurant);
}

@Override
public Collection<Restaurant> findByName(String name) throws
Exception {
    return restaurantRepository.findByName(name);
}

@Override
public void update(Restaurant restaurant) throws Exception {
    restaurantRepository.update(restaurant);
}

@Override
public void delete(String id) throws Exception {
    restaurantRepository.remove(id);
}

@Override
public Entity findById(String restaurantId) throws Exception {
    return restaurantRepository.get(restaurantId);
}

@Override
public Collection<Restaurant> findByCriteria(Map<String,
ArrayList<String>> name) throws Exception {
    throw new UnsupportedOperationException("Not supported yet.");
//若要更改生成的方法体，选择 Tools | Templates。
}
}
```

存储库类

`RestaurantRepository` 接口定义了两种新方法：`containsName` 和 `findByName` 方法。它还扩展了 `Repository` 接口：

```
public interface RestaurantRepository<Restaurant, String> extends
    Repository<Restaurant, String> {

    boolean containsName(String name) throws Exception;

    Collection<Restaurant> findByName(String name) throws Exception;
}
```

`Repository` 接口定义了三种方法：`and`、`remove` 和 `update`。它还扩展了 `ReadOnlyRepository` 接口：

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

`ReadOnlyRepository` 接口定义包含 `get` 和 `getAll` 方法，分别返回布尔值、实体和实体的集合。如果你想要只公开只读存储库的抽象，它非常有用：

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

Spring 框架使用 `@Repository` 注解来定义实现存储库的存储库 bean。在

RestaurantRepository的例子中,可以看到有一个映射被用来代替实际的数据库实现。这样,只在内存中保存所有的实体。因此,当我们启动服务时,发现在内存中只有两间餐馆。我们可以使用 JPA 来实现数据库持久性,这是用于生产的实现的一般做法:

```
@Repository("restaurantRepository")
public class InMemRestaurantRepository implements RestaurantRepository<Restaurant, String> {
    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Big-O Restaurant", "1", null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("O Restaurant", "2", null);
        entities.put("2", restaurant);
    }

    @Override
    public boolean containsName(String name) {
        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
            //异常处理程序
        }
        return false;
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }
}
```

```
}

@Override
public void update(Restaurant entity) {
    if (entities.containsKey(entity.getId())) {
        entities.put(entity.getId(), entity);
    }
}

@Override
public Collection<Restaurant> findByName(String name) throws
Exception {
    Collection<Restaurant> restaurants = new ArrayList();
    int noOfChars = name.length();
    entities.forEach((k, v) -> {
        if (v.getName().toLowerCase().contains(name.subSequence(0,
noOfChars))) {
            restaurants.add(v);
        }
    });
    return restaurants;
}

@Override
public boolean contains(String id) {
    throw new UnsupportedOperationException("Not supported yet.");
//若要更改生成的方法体，选择 Tools | Templates。
}

@Override
public Entity get(String id) {
    return entities.get(id);
}

@Override
public Collection<Restaurant> getAll() {
    return entities.values();
}
}
```

实体类

Restaurant 实体扩展了 BaseEntity，它的定义如下：

```
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();

    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, name: {}, capacity: {}",
this.getId(), this.getName(), this.getCapacity()));
        return sb.toString();
    }
}
```

 因为我们为实体的定义使用 POJO 类，所以在很多情况下不需要创建一个值对象。这个思路是，不应在实体之间保持对象的状态。

Table 实体扩展了 BaseEntity，它的定义如下：

```
public class Table extends BaseEntity<BigInteger> {
    private int capacity;
```

```
public Table(String name, BigInteger id, int capacity) {
    super(id, name);
    this.capacity = capacity;
}
public void setCapacity(int capacity) {
    this.capacity = capacity;
}
public int getCapacity() {
    return capacity;
}
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(String.format("id: {}, name: {}", this.getId(),
this.getName()));
    sb.append(String.format("Tables: {}" +
Arrays.asList(this.getTables())));
    return sb.toString();
}
}
```

Entity 抽象类定义如下：

```
public abstract class Entity<T> {

    T id;
    String name;

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

```
}

public void setName(String name) {
    this.name = name;
}

}
```

BaseEntity 抽象类的定义如下，它扩展了 Entity 抽象类：

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    private boolean isModified;
    private String name;

    public BaseEntity(T id, String name) {
        this.id = id;
        this.name = name;
    }

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public boolean isIsModified() {
        return isModified;
    }

    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }

    public String getName() {
```

```

        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}

```

预订和用户服务

我们可以使用 RestaurantService 实现来开发预订（Booking）和用户（User）服务。User 服务能提供与 CRUD 操作有关的用户资源的端点。Booking 服务可以提供与 CRUD 操作和餐桌位子的可用性相关的预订资源的端点。可以在 Packt 网站上找到这些服务的示例代码。

注册和发现服务（Eureka 服务）

Spring Cloud 提供了对 Netflix Eureka 最先进的支持，这是一种服务注册和发现工具。所有由你执行的服务都被 Eureka 服务列出和发现，这是从你的服务项目内的 Eureka 客户端 Spring 配置中读取的。

它需要如下所示的 Spring Cloud 依赖项和在 pom.xml 中的启动类，其中包含 @EnableEurekaApplication 注解：

Maven 依赖项：

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>

```

启动类：

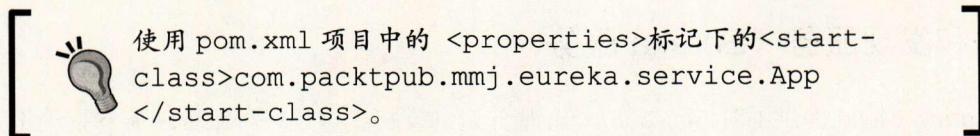
启动类 App 将只使用@EnableEurekaApplication 类注解来无缝地运行 Eureka 服务：

```
package com.packtpub.mmj.eureka.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```



Spring 的配置：

Eureka 服务也需要下列 Spring 配置信息用于 Eureka 服务器配置(src/main/resources/application.yml):

```
server:
  port: ${vcap.application.port:8761} # HTTP 端口

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

类似于 Eureka 服务器，每个 OTRS 服务还应包含 Eureka 客户端配置，以便可以建立 Eureka 服务器和客户端之间的连接。没有这一点，是不可能注册和发现服务的。

Eureka 客户端：你的服务可以使用以下的 spring 配置来配置 Eureka 服务器：

```
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka/
```

执行

为了查看我们的代码如何工作，需要首先生成它，然后执行它。我们将使用 Maven clean package 来建立服务 JAR。

现在若要执行这些服务 JAR 文件，只需从服务主目录执行下面的命令：

```
java -jar target/<service_jar_file>
```

例如：

```
java -jar target/restaurant-service.jar  
java -jar target/eureka-service.jar
```

测试

若要启用测试，需要在 pom.xml 中添加以下依赖项：

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
</dependency>
```

为了测试 RestaurantController，已添加了下列文件：

- RestaurantControllerIntegrationTests，使用 @SpringApplicationConfiguration 注解来选择 Spring Boot 使用的相同配置：

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@SpringApplicationConfiguration(classes = RestaurantApp.class)
public class RestaurantControllerIntegrationTests extends
    AbstractRestaurantControllerTests {
}
```

- 一个抽象类来编写我们的测试：

```
public abstract class AbstractRestaurantControllerTests {

    protected static final String RESTAURANT = "1";
    protected static final String RESTAURANT_NAME = "Big-O Restaurant";

    @Autowired
    RestaurantController restaurantController;

    @Test
    public void validRestaurantById() {
        Logger.getGlobal().info("开始validRestaurantById 测试");
        ResponseEntity<Entity> restaurant = restaurantController.
            findById(RESTAURANT);

        Assert.assertEquals(HttpStatus.OK, restaurant.
            getStatusCode());
        Assert.assertTrue(restaurant.hasBody());
        Assert.assertNotNull(restaurant.getBody());
        Assert.assertEquals(RESTAURANT, restaurant.getBody().
            getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.getBody().
            getName());
        Logger.getGlobal().info("结束validRestaurantById 测试");
    }

    @Test
    public void validRestaurantByName() {
        Logger.getGlobal().info("开始validRestaurantByName 测试");
        ResponseEntity<Collection<Restaurant>> restaurants =
            restaurantController.findByName(RESTAURANT_NAME);
```

```

        Logger.getGlobal().info("正在 validAccount 测试");

        Assert.assertEquals(HttpStatus.OK, restaurants.
getStatusCode());
        Assert.assertTrue(restaurants.hasBody());
        Assert.assertNotNull(restaurants.getBody());
        Assert.assertFalse(restaurants.getBody().isEmpty());
        Restaurant restaurant = (Restaurant) restaurants.
getBody().toArray()[0];
        Assert.assertEquals(RESTAURANT, restaurant.getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.
getName());
        Logger.getGlobal().info("结束validRestaurantByName 测试");
    }

@Test
public void validAdd() {
    Logger.getGlobal().info("开始validAdd 测试");
    RestaurantVO restaurant = new RestaurantVO();
    restaurant.setId("999");
    restaurant.setName("测试Restaurant");

    ResponseEntity<Restaurant> restaurants =
restaurantController.add(restaurant);
    Assert.assertEquals(HttpStatus.CREATED, restaurants.
getStatusCode());
    Logger.getGlobal().info("结束validAdd 测试");
}
}

```

- 最后，RestaurantControllerTests 扩展了以前创建的抽象类，也创建了 RestaurantService 和 RestaurantRepository 实现：

```

public class RestaurantControllerTests extends
AbstractRestaurantControllerTests {

    protected static final Restaurant restaurantStaticInstance =

```

```
new Restaurant(RESTAURANT,
    RESTAURANT_NAME, null);

protected static class TestRestaurantRepository implements Res
taurantRepository<Restaurant, String> {

    private Map<String, Restaurant> entities;

    public TestRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Big-O Restaurant", "1",
null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("O Restaurant", "2", null);
        entities.put("2", restaurant);
    }

    @Override
    public boolean containsName(String name) {
        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
            //异常处理程序
        }
        return false;
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }
}
```

```
}

@Override
public void update(Restaurant entity) {
    if (entities.containsKey(entity.getId())) {
        entities.put(entity.getId(), entity);
    }
}

@Override
public Collection<Restaurant> findByName(String name)
throws Exception {
    Collection<Restaurant> restaurants = new ArrayList();
    int noOfChars = name.length();
    entities.forEach((k, v) -> {
        if (v.getName().toLowerCase().contains(name.
subSequence(0, noOfChars))) {
            restaurants.add(v);
        }
    });
    return restaurants;
}

@Override
public boolean contains(String id) {
    throw new UnsupportedOperationException("Not supported
yet."); //若要更改生成的方法体，选择 Tools | Templates。
}

@Override
public Entity get(String id) {
    return entities.get(id);
}

@Override
public Collection<Restaurant> getAll() {
    return entities.values();
}
```

```
}

protected TestRestaurantRepository testRestaurantRepository =
new TestRestaurantRepository();
protected RestaurantService restaurantService = new Restaurant
ServiceImpl(testRestaurantRepository);

@Before
public void setup() {
    restaurantController = new RestaurantController(restaurant
Service);
}
}
```

参考资料

- *RESTful Java Patterns and Best Practices* (REST 式的 Java 模式和最佳实践), Bhakti Mehta 著, Packt 出版社: <https://www.packtpub.com/application-development/restful-javapatterns-and-best-practices>
- *Spring Cloud*: <http://cloud.spring.io/>
- *Netflix Eureka*: <https://github.com/netflix/eureka>

小结

本章我们学习了如何在微服务中使用领域驱动设计模型。运行演示应用程序后, 我们可以看到如何独立地开发、部署, 并测试每个微服务。可以很方便地使用 Spring Cloud 来创建微服务。同时, 我们也探讨如何使用 Eureka 注册和发现 Spring Cloud 组件。

在下一章中, 我们将学习在诸如 Docker 的容器中部署微服务, 还将了解使用其他 Java 客户端和其他工具来测试微服务。

5

部署和测试

这一章将会解释如何用不同的形式，包括独立部署和使用诸如 Docker 的容器来部署微服务，还将演示如何用 Docker 把我们的示例项目部署到云服务如 AWS 上。在实现 Docker 之前，我们将首先探索微服务的其他相关因素，如负载均衡和边缘服务器，你也将了解使用不同的 REST 客户端，如 RestTemplate、Netflix Feign 等来测试微服务。

在这一章，我们将介绍以下主题：

- 使用 Netflix OSS 的微服务架构概述
- 微服务的负载均衡
- 边缘服务器
- 断路器和监控
- 使用容器部署微服务
- 使用 Docker 容器对微服务进行集成测试

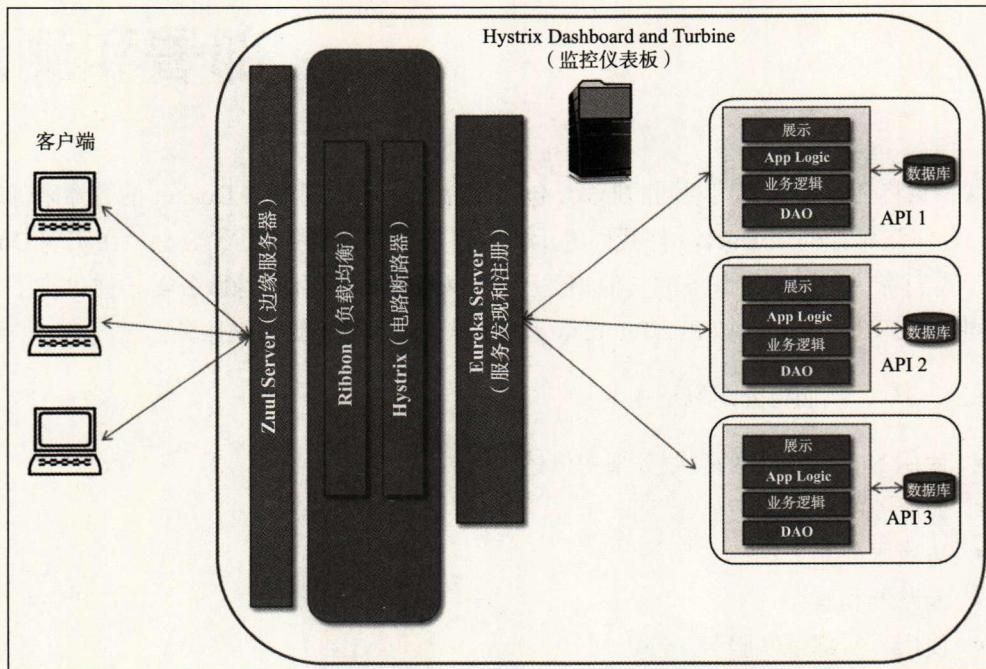
使用 Netflix OSS 的微服务架构概述

Netflix 是微服务架构中的先锋。它们是第一个成功在很大程度上实现微服务架构的组织，它们也有助于提高其受欢迎程度，并通过 **Netflix 开放源码软件中心 (Open Source Software Center, OSS)** 开源它们的大多数微服务工具，为推广微服务做出了巨大的贡献。

按照 Netflix 博客的描述，当 Netflix 开发自己的平台时，它们使用 Apache Cassandra 的数据存储，这是来自 Apache 的一种开源工具。它们开始通过修复和优化扩展为 Cassandra

做贡献。这使得 Netflix 看到用开放源码软件中心这个名称来发布 Netflix 项目的好处。

Spring 借此机会，把许多 Netflix 开放源码软件项目，如 Zuul、Ribbon、Hystrix、Eureka Server 和 Turbine，都集成到 Spring Cloud 中。这是 Spring Cloud 能够为开发可用于生产的微服务提供现成平台的原因之一。现在，让我们看看几个重要的 Netflix 工具以及它们是如何适应微服务架构的。



微服务架构关系图

可以在上图中看到，对于每个微服务的做法，我们都有与它关联的 Netflix 工具。我们可以通过下面的映射来理解它。除了 Eureka 已在上一章中详细阐述了，其他工具的详细信息都在本章各节进行探讨。

- **边缘服务器：**我们使用 Netflix Zuul Server 作为边缘服务器。
- **负载均衡：**Netflix Ribbon 用于负载均衡。
- **电路断路器：**Netflix Hystrix 用作电路断路器和帮助保持系统运行。
- **服务发现和注册：**Netflix Eureka 服务器用于服务发现和注册。

- 监控仪表板：Hystrix 仪表板与 Netflix Turbine 结合用于微服务监控。它提供了一个仪表板来检查运行中的微服务的健康状况。

负载均衡

如果要以速度快、容量利用率最大化的方式为请求提供服务，就需要实现负载均衡，这样可以确保没有服务器的请求超负荷。如果一台服务器出现故障，负载均衡器也会将请求重定向到其余的主机服务器上。在微服务架构中，微服务可以为内部或外部的请求提供服务，在此基础上，我们可以有两种类型的负载均衡——客户端负载均衡和服务器端负载均衡。

客户端的负载均衡

微服务需要使用进程间通信，以便服务可以互相交流。Spring Cloud 使用 Netflix Ribbon 承担负载均衡器这个至关重要的角色，并可以处理 HTTP 和 TCP 客户端的负载。Ribbon 可用于云平台，并提供内置的故障恢复能力。Ribbon 还允许使用多个和可插拔的负载均衡规则，它集成了客户端和负载均衡器。

在上一章，我们添加了 Eureka 服务器。在 Spring Cloud 中，Ribbon 在默认情况下与 Eureka 服务器集成。这种集成提供了以下功能：

- 在使用 Eureka 服务器时，不需要对要发现的远程服务器 URL 进行硬编码。这是一个突出的优势，尽管你仍然可以根据需要在 application.yml 中使用已配置的服务器列表 (*listOfServers*)。
- 服务器列表由 Eureka 服务器来填充。Eureka 服务器用 DiscoveryEnabledNIWSServerList 重写 ribbonServerList。
- 找出服务器是否在运行的请求被委托给 Eureka。

DiscoveryEnabledNIWSServerList 接口用于替代 Ribbon 的 IPing。

Spring Cloud 中有使用 Ribbon 的不同客户端，如 **RestTemplateor** 或 **FeignClient**。这些客户端允许微服务间相互通信。使用 Eureka 服务器时，客户端使用实例 ID 代替主机名和端口用于 HTTP 向服务实例发出调用。客户端把服务 ID 传给 Ribbon，Ribbon 然后使用负载均衡器获取 Eureka 服务器实例。

如果在 Eureka 中有可用的服务的多个实例，如下面的屏幕截图所示，Ribbon 基于负载均衡算法，只为请求获取一个实例：

Instances currently registered with Eureka				
Application	AMIs	Availability		Status
		Zones	Status	
RESTAURANT-SERVICE	n/a (2)	{2}	UP (2) - SOUSHARM-IN:restaurant-service:5b034f31fd44c9ff6dd5c5fb1d4c83d7}, SOUSHARM-IN:restaurant-service:707b060d8d02e3516f3fde3c86c858d1}	
ZUUL-SERVER	n/a (1)	{1}	UP (1) - SOUSHARM-IN:zuul-server:9094e5aae179efe903061d827e21e167}	

多个服务注册——餐馆服务

我们可以使用 `DiscoveryClient` 来查找 Eureka 服务器中可用的服务的所有实例，如下面的代码所示。`DiscoveryClientSample` 类的 `getLocalServiceInstance()` 方法返回 Eureka 服务器中所有可用的本地服务实例。

DiscoveryClient 示例：

```
@Component
class DiscoveryClientSample implements CommandLineRunner {

    @Autowired
    private DiscoveryClient;

    @Override
    public void run(String... strings) throws Exception {
        //打印发现客户端的描述
        System.out.println(discoveryClient.description());
        //获得餐馆服务实例并打印它的信息
        discoveryClient.getInstances("restaurant-service").
        forEach((ServiceInstance serviceInstance) -> {
            System.out.println(new StringBuilder("Instance -->
").append(serviceInstance.getServiceId())
                .append("\nServer: ").append(serviceInstance.
getHost()).append(":").append(serviceInstance.getPort())
                .append("\nURI: ").append(serviceInstance.
getUri()).append("\n\n"));
        });
    }
}
```

```

    });
}
}
}

```

在执行时，此代码将打印以下信息。它显示了餐馆服务的两个实例：

```

Spring Cloud Eureka Discovery Client
Instance: RESTAURANT-SERVICE
Server: SOUSHARM-IN:3402
URI: http://SOUSHARM-IN:3402
Instance --> RESTAURANT-SERVICE
Server: SOUSHARM-IN:3368
URI: http://SOUSHARM-IN:3368

```

下面的示例展示如何使用这些客户端。可以看到，在两个客户端中，都用服务名 `restaurant-service` 来替代服务的主机名和端口。这些客户端调用 `/v1/restaurants` 来获取在名称查询参数中所包含的名字的餐馆列表：

Rest 模板示例：

```

@Override
public void run(String... strings) throws Exception {
    ResponseEntity<Collection<Restaurant>> exchange
        = this.restTemplate.exchange(
            "http://restaurant-service/v1/restaurants?name=o",
            HttpMethod.GET,
            null,
            new ParameterizedTypeReference<Collection<Restaurant>>() {
            },
            ( "restaurants" );
    exchange.getBody().forEach((Restaurant restaurant) -> {
        System.out.println(new StringBuilder("\n\n\n[ ")
            .append(restaurant.getId())
            .append(" ")
            .append(restaurant.getName())
            .append("]")));
    });
}

```

FeignClient 示例：

```
@Component
```

```
class FeignSample implements CommandLineRunner {

    @Autowired
    private RestaurantClient restaurantClient;

    @Override
    public void run(String... strings) throws Exception {
        this.restaurantClient.getRestaurants("o").forEach((Restaurant
restaurant) -> {
            System.out.println(restaurant);
        });
    }
}

@FeignClient("restaurant-service")
interface RestaurantClient {

    @RequestMapping(method = RequestMethod.GET, value = "/v1/restaurants")
    Collection<Restaurant> getRestaurants(@RequestParam("name") String
name);
}
```

上述所有示例将都打印以下输出：

```
[ 1 Big-O Restaurant]
[ 2 O Restaurant]
```

服务器端的负载均衡

在实现客户端的负载均衡后，定义服务器端的负载均衡是重要的。此外，从微服务架构的角度来看，定义我们的 OTRS 应用程序的路由机制是很重要的。例如，/可能被映射到我们的 UI 应用程序，/restaurantapi 被映射到餐馆服务，而/userapi 被映射到用户服务。

我们将使用 Netflix Zuul Server 作为我们的边缘服务器。Zuul 是一个基于 JVM 的路由器和服务器端的负载均衡器。Zuul 支持用任何 JVM 语言来编写规则和过滤条件并具有内置的 Java 和 Groovy 的支持。

外部世界（UI 和其他客户端）调用边缘服务器，它使用在 application.yml 中定义的路由，调用内部服务并提供响应。如果你认为它充当代理服务器、承担内部网络的网关，并为定义和配置的路由调用内部服务，你的猜测是正确的。

通常情况下，推荐为所有请求提供一个单独的边缘服务器。然而，少数几家公司为每个客户端使用单独的边缘服务器，以便扩展。例如，Netflix 为每种设备类型使用专用的边缘服务器。

在下一章，当我们配置并实现微服务的安全性时，也将使用边缘服务器。

在 Spring Cloud 中配置和使用边缘服务器非常简单，只需要采取以下步骤：

1. 在 pom.xml 中定义 Zuul 服务器依赖项：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

2. 在应用程序类中使用 @EnableZuulProxy 注解。它还会在内部使用 @EnableDiscoveryClient：因此它也会自动注册到 Eureka 服务器。可以在最后一个图形：多个服务注册——餐馆服务中找到已注册的 Zuul 服务器。

3. 在 application.yml 中更新 Zuul 配置，如下所示：

- zuul:ignoredServices：这跳过服务的自动添加。我们可以在这里定义服务 ID 模式。* 表示我们忽略了所有的服务。在以下示例中，除了 restaurant-service 外的所有服务都将被忽略。
- Zuul.routes：这包含定义 URI 模式的 path 属性。在这里，/restaurantapi 使用 serviceId 被映射到餐馆服务。serviceId 表示在 Eureka 服务器中的服务。如果不使用 Eureka 服务器，可以使用服务的 URL 来替换。我们也用 stripPrefix 属性来剥离前缀(/restaurantapi)，由此导致在调用此服务时，/restaurantapi/v1/estaurants/1 调用将转换为/v1/restaurants/1：

```
application.yml
info:
```

```
component: Zuul Server
# Spring的属性
spring:
  application:
    name: zuul-server #服务注册在这个名字下

endpoints:
  restart:
    enabled: true
  shutdown:
    enabled: true
  health:
    sensitive: false

zuul:
  ignoredServices: "*"
  routes:
    restaurantapi:
      path: / restaurantapi/**
      serviceId: restaurant-service
      stripPrefix: true

server:
  port: 8765

# 发现服务器访问
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 3
    metadataMap:
      instanceId: ${vcap.application.instance_id}:${spring.
application.name}:${spring.application.instance_id:${random.
value}}}
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    fetchRegistry: false
```

让我们看看工作中的边缘服务器。首先，我们调用部署在端口 3402 上的餐馆服务，如下图所示。

```

http://localhost:3402/v1/restaurants?name=o
GET

Send Preview Add to collection

Body Headers (4) STATUS 200 OK TIME 46 ms

Pretty Raw Preview JSON XML

1 [
2   {
3     "id": "1",
4     "name": "Big-O Restaurant",
5     "isModified": false,
6     "tables": null
7   },
8   {
9     "id": "2",
10    "name": "O Restaurant",
11    "isModified": false,
12    "tables": null
13  }
14 ]
  
```

直接的餐馆服务调用

然后，我们会使用部署在端口 8765 上的边缘服务器调用相同的服务。可以看到，调用 /v1/restaurants?name=o 用到了 /restaurantapi 前缀，并且它给出了相同的结果。

```

http://localhost:8765/restaurantapi/v1/restaurants?name=o
GET

Send Preview Add to collection

Body Headers (5) STATUS 200 OK TIME 604 ms

Pretty Raw Preview JSON XML

1 [
2   {
3     "id": "1",
4     "name": "Big-O Restaurant",
5     "isModified": false,
6     "tables": null
7   },
8   {
9     "id": "2",
10    "name": "O Restaurant",
11    "isModified": false,
12    "tables": null
13  }
14 ]
  
```

使用边缘服务器的餐馆服务调用

电路断路器与监控

总体而言，电路断路器是：

在电路中作为一项安全措施来停止电流流动的一种自动装置。

在微服务开发中，使用了相同的概念，这就是称为电路断路器的设计模式。它跟踪 Eureka 服务器等外部服务、餐馆服务等 API 服务的可用性，并防止服务使用者在任何不可用的服务上执行任何操作。

这是微服务架构的另一个重要方面，当服务不能响应服务使用者的调用时的一项安全措施（故障安全机制）——电路断路器。

我们将使用 Netflix Hystrix 作为电路断路器。在发生故障时（例如由于通信错误或超时），它调用服务使用者的内部回退方法，它嵌入在其服务的使用者中执行。在下一节中，你会看到实现此功能的代码。

Hystrix 打开电路并在服务多次无法响应时快速断开，直到此服务再次可用时再打开。你一定会想，如果 Hystrix 打开电路，那么它如何知道此服务是可用的呢？它也例外地允许一些请求来调用此服务。

使用 Hystrix 的回退方法

为了实现回退方法，需要采取三个步骤：

1. 启用断路器：使用其他服务的微服务的主类应使用 @EnableCircuitBreaker 注解。例如，如果一个用户服务想要得到用户已预订了餐桌的餐馆的详细信息：

```
@SpringBootApplication
@EnableCircuitBreaker
@ComponentScan({"com.packtpub.mmj.user.service", "com.packtpub.
mmj.common"})
public class UsersApp {
```

2. 配置回退方法：使用 @HystrixCommand 注解来配置 fallbackMethod：

```
@HystrixCommand(fallbackMethod = "defaultRestaurant")
public ResponseEntity<Restaurant> getRestaurantById(int
```

```

restaurantId) {

    LOG.debug("Get Restaurant By Id with Hystrix protection");

    URI uri = util.getServiceUrl("restaurant-service");

    String url = uri.toString() + "/v1/restaurants/" +
restaurantId;
    LOG.debug("Get Restaurant By Id URL: {}", url);

    ResponseEntity<Restaurant> response = restTemplate.
getForEntity(url, Restaurant.class);
    LOG.debug("Get Restaurant By Id http-status: {}", response.
getStatusCode());
    LOG.debug("GET Restaurant body: {}", response.getBody());

    Restaurant restaurant = response.getBody();
    LOG.debug("Restaurant ID: {}", restaurant.getId());

    return serviceHelper.createOkResponse(restaurant);
}

```

3. 定义回退方法：处理故障和安全执行步骤的一个方法：

```

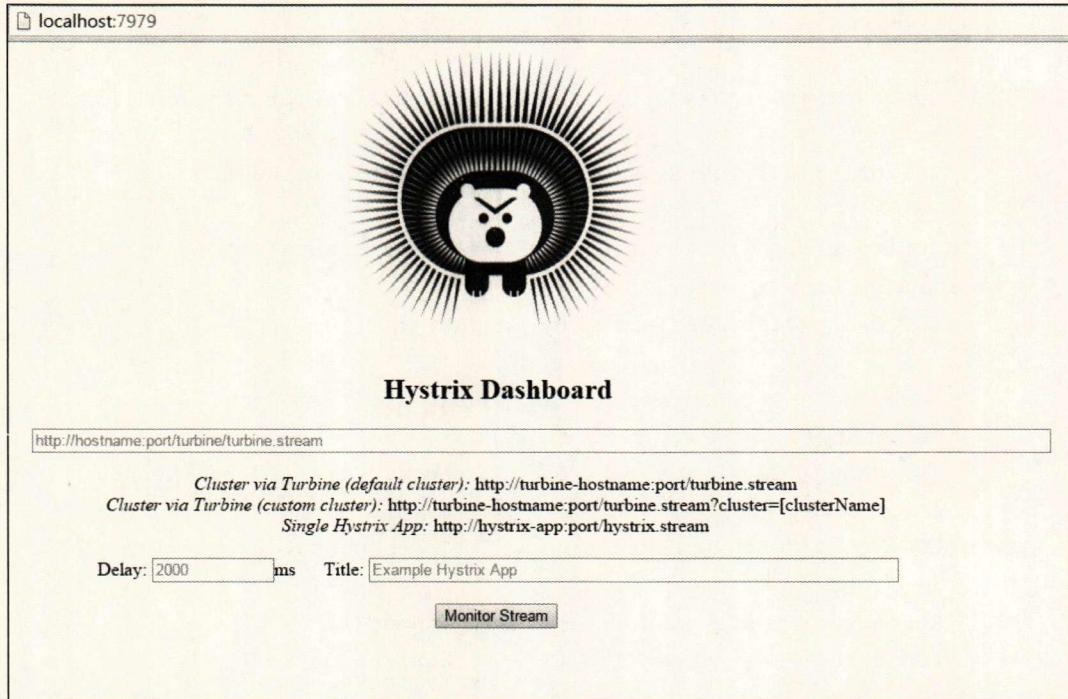
public ResponseEntity<Restaurant> defaultRestaurant(int
restaurantId) {
    LOG.warn("餐馆服务回退方法正在使用");
    return serviceHelper.createResponse(null, HttpStatus.BAD_
GATEWAY);
}

```

这些步骤足以对服务调用进行故障保护，并向服务使用者返回更为合适的响应。

监控

Hystrix 提供 web 用户界面的仪表板，它提供漂亮的电路断路器图形界面。



默认Hystrix仪表板

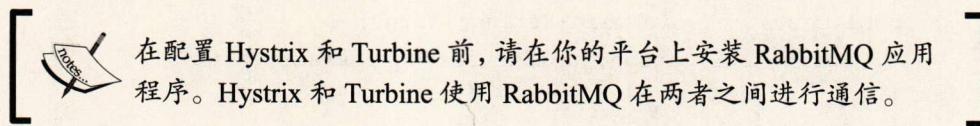
Netflix Turbine 是一个 web 应用程序，它连接到集群中 Hystrix 应用程序的实例并会实时（每隔 0.5 秒更新）聚合信息。Turbine 使用一个称为 Turbine 流的流来提供信息。

如果把 Hystrix 与 Netflix Turbine 结合使用，那么可以在 Hystrix 的仪表板上从所有 Eureka 服务器获得信息。这使你获得与电路断路器有关的所有信息的全貌。

要结合使用 Turbine 和 Hystrix，只需在之前显示的第一个文本框中键入 Turbine URL `http://localhost:8989/turbine.stream` (在 `application.yml` 中为 Turbine 服务器配置的端口是 8989)，并点击 **Monitory Stream** (监控流)。

Netflix Hystrix 和 Turbine 使用 RabbitMQ 这个开源消息队列软件。RabbitMQ 使用高级消息队列协议 (**Advance Messaging Queue Protocol, AMQP**) 工作。可以在这个软件中定义队列，应用程序可以借助它建立连接，并通过它传递消息。一条消息可以包含任何类型的信息。可以在 RabbitMQ 队列中存储消息，直到接收应用程序连接到它并接收此消息为止 (将消息从队列中取走)。

Hystrix 使用 RabbitMQ 来发送馈送给 Turbine 的度量数据。



设置 Hystrix 仪表板

我们将为 Hystrix 服务器添加 Maven 新依赖项, dashboard-server。像其他环境一样, 在 Spring Cloud 中配置和使用 Hystrix 仪表板是很简单的, 只需要按照下列步骤操作:

1. 在 pom.xml 中定义 Hystrix 仪表板依赖项:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</
    artifactId>
</dependency>
```

2. 使用主 Java 类中的 @EnableHystrixDashboard 注解完成了全部工作。我们也将使用 @Controller 把来自根的请求转发到 Hystrix, 如下所示:

```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class DashboardApp extends SpringBootServletInitializer {

    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplication
Builder application) {
        return application.sources(DashboardApp.class).web(true);
```

```
}

public static void main(String[] args) {
    SpringApplication.run(DashboardApp.class, args);
}
}
```

3. 在 application.yml 中更新 Dashboard 应用程序配置，如下所示：

```
application.yml
# Hystrix 仪表板属性
spring:
  application:
    name: dashboard-server

  endpoints:
    restart:
      enabled: true
    shutdown:
      enabled: true

  server:
    port: 7979

  eureka:
    instance:
      leaseRenewalIntervalInSeconds: 3
      metadataMap:
        instanceId: ${vcap.application.instance_id}:${spring.
          application.name}:${spring.application.instance_id:${random.
          value}}}

    client:
      # 来自org.springframework.cloud的默认值。
  netflix.eureka.EurekaClientConfigBean
    registryFetchIntervalSeconds: 5
    instanceInfoReplicationIntervalSeconds: 5
```

```

initialInstanceInfoReplicationIntervalSeconds: 5
serviceUrl:
  defaultZone: http://localhost:8761/eureka/
fetchRegistry: false

logging:
  level:
    ROOT: WARN
    org.springframework.web: WARN

```

设置 Turbine

我们会为 Turbine 再创建一个 Maven 依赖项。Hystrix 仪表板应用程序运行时，它会像前面的默认 *Hystrix* 仪表板截图那样显示。

现在，我们将使用以下步骤配置 Turbine 服务器：

1. 在 pom.xml 中定义 Turbine 服务器依赖项：

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-turbine-amqp</artifactId>
</dependency>

```

2. 在应用程序类中使用 @EnableTurbineAmqp 注解，如下所示。我们也定义了一个 bean，它会返回 RabbitMQ 连接工厂：

```

@SpringBootApplication
@EnableTurbineAmqp
@EnableDiscoveryClient
public class TurbineApp {

  private static final Logger LOG = LoggerFactory.
  getLogger(TurbineApp.class);

  @Value("${app.rabbitmq.host:localhost}")
  String rabbitMQHost;

```

```
@Bean
public ConnectionFactory connectionFactory() {
    LOG.info("Creating RabbitMQHost ConnectionFactory for
host: {}", rabbitMQHost);
    CachingConnectionFactory cachingConnectionFactory = new
CachingConnectionFactory(rabbitMQHost);
    return cachingConnectionFactory;
}

public static void main(String[] args) {
    SpringApplication.run(TurbineApp.class, args);
}
}
```

3. 在 application.yml 中更新 Turbine 配置，如下所示：

```
server:port: Turbine HTTP使用的主端口
management:port: Turbine执行器端点的端口
application.yml
spring:
  application:
    name: turbine-server

  server:
    port: 8989

  management:
    port: 8990

PREFIX:

endpoints:
  restart:
    enabled: true
  shutdown:
    enabled: true
```

```
eureka:  
  instance:  
    leaseRenewalIntervalInSeconds: 10  
  client:  
    registryFetchIntervalSeconds: 5  
    instanceInfoReplicationIntervalSeconds: 5  
    initialInstanceInfoReplicationIntervalSeconds: 5  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka/  
  
logging:  
  level:  
    root: WARN  
    com.netflix.discovery: 'OFF'
```



请注意前面的步骤始终使用默认配置创建各自的服务器。如果需要，你可以用特定的设置重写默认配置。

使用容器部署微服务

阅读第 1 章后，你可能已经了解了有关 Docker 的要点。

Docker 容器提供轻量级的运行时环境，它由虚拟机的核心功能和一个称为 Docker 映像的隔离的操作系统服务组成。Docker 使得包装和执行微服务更加容易、顺畅。每个操作系统可以有多个 Docker，并且每个 Docker 都可以运行多个应用程序。

安装和配置

如果你不使用 Linux 操作系统，那么 Docker 需要一台虚拟化的服务器。可以安装 VirtualBox 或类似的工具，如 Docker 工具箱来使它为你工作。Docker 安装页面提供了有关它的详细信息，并告诉你如何去做。所以，可在 Docker 的网站上阅读 Docker 安装指南。

可以按照在 <https://docs.docker.com/engine/installation/> 中给出的说明，基于你的平台安装 Docker。

DockerToolbox 1.9.1f 是写作时可用的最新版本。这是我们使用的版本。

具有 4 GB 内存的 Docker 机器

默认情况下创建的机器都有 2GB 的内存。我们将重新创建具有 4GB 内存的 Docker 机器：

```
docker-machine rm default
docker-machine create -d virtualbox --virtualbox-memory 4096 default
```

使用 Maven 构建 Docker 映像

有各种 Docker maven 插件可以用来构建映像：

- <https://github.com/rhuss/docker-maven-plugin>
- <https://github.com/alexec/docker-maven-plugin>
- <https://github.com/spotify/docker-maven-plugin>

可以根据你的选择使用任何一种。我发现由@rhuss 提供的 Docker Maven 插件最适合我们。它会定期更新，并且与其他的插件相比，有很多额外的功能。

在开始讨论 docker-maven-plugin 的配置前，我们需要介绍 application.yml 中的 Docker Spring 配置文件。在为各种平台构建服务时，它将使我们的工作更容易。我们需要配置以下四个属性：

- 我们会使用确定为 Docker 的 Spring 配置文件。
- 嵌入式 Tomcat 的端口之间不会有冲突，因为服务将在自己各自的容器内执行。我们现在可以使用端口 8080。
- 我们偏爱在 Eureka 中使用我们的服务的 IP 地址。因此，Eureka 实例属性 preferIpAddress 将被设置为 true。
- 最后，我们将在 serviceUrl:defaultZone 中使用 Eureka 服务器主机名。

为了把 Spring 配置文件添加到项目中，需要在 application.yml 中现有的内容后添加以下行：

```
---  
#用于在Docker容器中部署  
spring:  
    profiles: docker  
  
server:  
    port: 8080  
  
eureka:  
    instance:  
        preferIpAddress: true  
    client:  
        serviceUrl:  
            defaultZone: http://eureka:8761/eureka/
```

在生成 Docker 容器 JAR 时，我们还将在 `pom.xml` 中添加以下代码来激活 Spring 配置文件 Docker（这将使用先前定义的属性创建 JAR，例如端口：8080）。

```
<profiles>  
    <profile>  
        <id>docker</id>  
        <properties>  
            <spring.profiles.active>docker</spring.profiles.active>  
        </properties>  
    </profile>  
</profiles>
```

我们构建服务时只需要使用 Maven docker 配置文件，如下所示：

```
mvn -P docker clean package
```

上面的命令将生成包含 Tomcat 的 8080 端口的 service JAR，并将在 Eureka 服务器上用主机名 Eureka 注册。

现在，让我们配置 `docker-maven-plugin` 生成包含餐馆微服务的映像。这个插件必须首先创建一个 `Dockerfile`。`Dockerfile` 是在两个地方配置的——在 `pom.xml` 和 `docker-assembly.xml` 中。我们将在 `pom.xml` 中使用如下插件配置：

```
<properties>
<!—对于 Docker hub 保留空白；对于本地Docker Registry使用 "localhost:5000/" -->
<docker.registry.name>localhost:5000/</docker.registry.name>
<docker.repository.name>${docker.registry.name}sourabhh /${project.
artifactId}</docker.repository.name>
</properties>
...
<plugin>
<groupId>org.jolokia</groupId>
<artifactId>docker-maven-plugin</artifactId>
<version>0.13.7</version>
<configuration>
<images>
<image>
<name>${docker.repository.name}:${project.version}</name>
<alias>${project.artifactId}</alias>

<build>
<from>java:8-jre</from>
<maintainer>sourabhh</maintainer>
<assembly>
<descriptor>docker-assembly.xml</descriptor>
</assembly>
<ports>
<port>8080</port>
</ports>
<cmd>
<shell>java -jar \
/maven/${project.build.finalName}.jar server \
/maven/docker-config.yml</shell>
</cmd>
</build>
<run>
<!-- To Do -->
</run>
</image>
```

```

</images>
</configuration>
</plugin>

```

以上的 Docker Maven 插件配置，创建了一个创建基于 JRE 8 (java:8-jre)的映像的 Dockerfile。这公开了端口 8080 和 8081。

接下来，我们将配置 docker-assembly.xml，它告诉插件应将哪些文件放入容器。它将被放置在 src/main/docker 目录下：

```

<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
  <id>${project.artifactId}</id>
  <files>
    <file>
      <source>{basedir}/target/${project.build.finalName}.jar</source>
      <outputDirectory></outputDirectory>
    </file>
    <file>
      <source>src/main/resources/docker-config.yml</source>
      <outputDirectory></outputDirectory>
    </file>
  </files>
</assembly>

```

上方的组装，把 service JAR 和 docker-config.yml 添加到生成的 Dockerfile 中。这个 Dockerfile 位于 target/docker/ 目录下。打开此文件，你将会发现类似于下面的内容：

```

FROM java:8-jre
MAINTAINER sourabhh
EXPOSE 8080
COPY maven /maven/
CMD java -jar \

```

```
/maven/restaurant-service.jar server \
/maven/docker-config.yml
```

前面的文件位于 `restaurant-service\target\docker\sousharm\restaurant-service\PACKT-SNAPSHOT\build`。Build 目录中还包含 maven 目录，其中包含在 `docker-assembly.xml` 中提到的所有东西。

让我们生成 Docker 映像：

```
mvn docker:build
```

此命令完成后，我们可以使用 Docker Images 或通过运行以下命令来验证本地存储库中的映像：

```
docker run -it -p 8080:8080 sourabhh/restaurant-service:PACKT-SNAPSHOT
```

使用`-it` 代替`-d` 在前台执行此命令。

使用 Maven 运行 Docker

若要使用 Maven 来执行 Docker 映像，我们需要在 `pom.xml` 中添加以下配置项。把 `<run>` 块放在我们在 `pom.xml` 文件 `docker-maven-plugin` 部分的 `image` 块下标记 *To Do* 的地方：

```
<properties>
  <docker.host.address>localhost</docker.host.address>
  <docker.port>8080</docker.port>
</properties>
...
<run>
  <namingStrategy>alias</namingStrategy>
  <ports>
    <port>${docker.port}:8080</port>
  </ports>
  <volumes>
    <bind>
      <volume>${user.home}/logs:/logs</volume>
    </bind>
```

```

</volumes>
<wait>
  <url>http://${docker.host.address}:${docker.port}/v1/
restaurants/1</url>
  <time>100000</time>
</wait>
<log>
  <prefix>${project.artifactId}</prefix>
  <color>cyan</color>
</log>
</run>

```

在这里，我们已经定义了运行餐馆服务的容器参数。我们映射了 Docker 容器的 8080 和 8081 端口到主机系统的端口，使我们能够访问此服务。同样，我们也已把容器的日志目录绑定到主机系统的<home>/logs 目录。

Docker Maven 插件通过轮询后台管理的 ping URL，直到它接收到一个回复，可以检测容器是否已完成启动。

请注意，如果你在 Windows 或者 Mac OS X 上使用 DockerToolbox 或 boot2docker，Docker 主机不是 localhost。可以通过执行 docker-machine ip default 检查 Docker 映像的 IP，它也在启动时显示。

Docker 容器已经准备就绪，用下面的命令来使用 Maven 启动它：

```
mvn docker:start .
```

使用 Docker 执行集成测试

启动和停止 Docker 容器可以通过将以下执行块绑定到在 pom.xml 中的 docker-maven-plugin 生命周期阶段来完成：

```

<execution>
  <id>start</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>build</goal>

```

如果你还没有设置 docker 本地寄存处，那么请首先做这个工作，以便问题较少或更顺畅地执行。

构建 docker 的本地寄存处：

```
docker run -d -p 5000:5000 --restart=always --name
registry registry:2
```



然后，对本地的映像执行 push 和 pull 命令：

```
docker push localhost:5000/sourabhh/restaurant-
service:PACKT-SNAPSHOT
docker-compose pull
```

最后，执行 docker-compose：

```
docker-compose up -d
```

一旦所有的微服务容器（服务和服务器）都被配置完成，我们就可以用单个命令启动所有 Docker 容器：

```
docker-compose up -d
```

这将启动在 Docker Compose 中配置的所有 Docker 容器。以下命令将列出它们：

```
docker-compose ps
```

Name	State	Ports	Command
onlinetablereservation5_eureka_1	Up	0.0.0.0:8761->8761/tcp	/bin/sh -c java -jar ...
onlinetablereservation5_restaurant-service_1	Up	0.0.0.0:8080->8080/tcp	/bin/sh -c java -jar ...

还可以使用以下命令检查 Docker 映像日志：

```
docker-compose logs
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.819 INFO 7 --- [pool-3-thread-1] com.netflix.discovery.DiscoveryClient : DiscoveryClient_RESTAURANT-SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce - Re-registering
apps/RESTAURANT-SERVICE
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.820 INFO 7 --- [pool-
```

```
3-thread-1] com.netflix.discovery.DiscoveryClient : DiscoveryClient_
RESTAURANT-SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dc3d86c858e520d3ce: registering
service...
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.917 INFO 7 --- [pool-
3-thread-1] com.netflix.discovery.DiscoveryClient : DiscoveryClient_
RESTAURANT-SERVICE/172.17
```

参考资料

以下链接提供详细信息：

- **Netflix Ribbon:** <https://github.com/Netflix/ribbon>
- **Netflix Zuul:** <https://github.com/Netflix/zuul>
- **RabbitMQ:** <https://www.rabbitmq.com/download.html>
- **Hystrix:** <https://github.com/Netflix/Hystrix>
- **Turbine:** <https://github.com/Netflix/Turbine>
- **Docker:** <https://www.docker.com/>

小结

在这一章，我们已经了解了各种微服务管理功能——负载均衡、边缘服务器（网关）、电路断路器和监控。通过这一章的学习，现在应该知道如何实现负载均衡和路由，我们也学到了如何设置边缘服务器和配置。故障保护机制是在这一章已经学会的另一个重要部分。利用 Docker 或任何其他容器，可以简化部署。我们还使用 Maven 构建来演示了 Docker 和集成。

从测试的角度，我们对服务的 Docker 映像执行集成测试，还探讨了编写 RestTemplate 和 Netflix Feign 等客户端的方法。

在下一章中，我们将学会从身份验证和授权方面实现微服务的安全性，也将探讨微服务安全性的其他方面。

6

实现微服务的安全性

正如你所知，微服务是我们部署在处所内或云基础设施中的组件，微服务可以提供 API 或 web 应用程序。我们的示例应用程序 OTRS 提供的是 API。这一章将侧重如何使用 Spring Security 和 Spring OAuth2 来实现这些 API 的安全性，还会重点介绍 OAuth 2.0 基础知识，我们会使用 OAuth 2.0 来保护 OTRS 的 API。有关 REST API 安全保护的更多理解，可以参考 Packt 出版的《*RESTful Java Web Services Security*》一书，也可以参考 Packt 出版的《*Spring Security [Video]*》来学习更多的信息。我们也会了解跨起源请求网站(Cross Origin Request Site)过滤器和跨站点脚本阻塞程序。

在这一章，我们将介绍以下主题：

- 启用安全套接字层（SSL）
- 身份验证和授权
- OAuth 2.0

启用安全套接字层

到目前为止，我们都在使用超文本传输协议（**Hyper Text Transfer Protocol, HTTP**）。HTTP 以纯文本形式传输数据，但以纯文本形式通过互联网传输数据，完全不是一个好主意。它方便了黑客的工作，使他们能够使用数据包嗅探器很容易地得到你的私人信息，如你的用户 ID、密码和信用卡详细信息。

我们肯定不想用户数据受到危害，所以我们将提供最安全的方式来访问我们的 web 应

用程序。因此，我们需要对最终用户和应用程序之间交换的信息进行加密。我们会使用安全套接字层（**Secure Socket Layer, SSL**）或传输安全层（**Transport Security Layer, TSL**）来加密数据。

SSL 是一种旨在为网络通信提供安全性（加密）的协议。HTTP 与 SSL 结合提供了 HTTP 的安全性实现，被称为安全超文本传输协议（**Hyper Text Transfer Protocol Secure**），或通过 SSL 的超文本传输协议（**Hyper Text Transfer Protocol over SSL, HTTPS**）。HTTPS 可以确保被交换数据的隐私和完整性得到保护，它还确保被访问网站的真实性。此安全性围绕着托管应用程序的服务器、最终用户的机器，以及第三方信任存储服务器之间分布签名的数字证书。让我们看看这一过程是如何发生的：

1. 最终用户使用 web 浏览器将请求发送到 web 应用程序，例如 `http://twitter.com`。
2. 在接收到请求后，服务器使用 HTTP 代码 302 将浏览器重定向到 `https://twitter.com`。
3. 最终用户的浏览器链接到 `https://twitter.com`，并且在响应中，服务器把其中包含数字签名的证书提供给最终用户的浏览器。
4. 最终用户的浏览器接收该证书，并将其发送到受信任的证书颁发机构（**Certificate Authority, CA**）进行验证。
5. 一旦证书获得根 CA 的验证，最终用户的浏览器和应用程序宿主服务器之间就建立了加密的通信。





尽管 SSL 用加密和 web 应用程序身份验证的方式来保证安全性，但它不能保障不受网络钓鱼和其他攻击。专业黑客可以对使用 HTTPS 发送的信息进行解密。

现在，复习过 SSL 的基本知识，让我们为示例 OTRS 的项目实现它。我们不需要为所有的微服务都实现 SSL。除了新的微服务，即我们将在本章介绍的进行身份验证和授权的安全服务以外，所有其他微服务将都使用我们的代理或边缘服务器，Zuul 服务器被外部环境访问。

首先，我们会在边缘服务器上设置 SSL。我们需要有在嵌入式 Tomcat 中启用 SSL 所需的密钥存储库。为了演示，我们会使用自签名的证书，使用 Java keytool 用下面的命令生成密钥存储库，也可以使用任何其他工具来完成这件事：

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext
san=dns:localhost -storepass password -validity 365 -keysize 2048
```

它要求提供诸如名称、详细地址、组织等信息（见下面的截图）。

```
C:\dev\workspace\ms\online-table-reservation-6>keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext
what is your first and last name?
[Unknown]: localhost
what is the name of your organizational unit?
[Unknown]: org unit
what is the name of your organization?
[Unknown]: org
what is the name of your City or Locality?
[Unknown]: city
what is the name of your State or Province?
[Unknown]: state
what is the two-letter country code for this unit?
[Unknown]: CN
Is CN=localhost, OU=org unit, O=org, L=city, ST=state, C=CN correct?
[no]: yes

Enter key password for <selfsigned>
      (RETURN if same as keystore password):
Re-enter new password:

C:\dev\workspace\ms\online-table-reservation-6>
```

Keytool生成密钥

应注意以下几点以确保自签名证书的功能正常：

- 使用-ext 来定义主题备用名称（**Subject Alternative Names, SAN**），还可以使用 IP（例如，san =ip:190.19.0.11）。早些时候，应用程序被部署到的机器的

主机名，被用作最常见的名称 **common name (CN)**。这个设置防止 No name matching localhost found (未找到匹配 localhost 的名称) 的 java.security.cert.CertificateException。

- 可以使用浏览器或 OpenSSL 下载证书。利用 keytool-importcert 命令将新生成的证书添加到位于活动的 JDK/JREhome 目录 jre/lib/security/cacerts 中的 cacerts 密钥库。请注意，cacerts 密钥库的默认密码是 changeit。运行以下命令：

```
keytool -importcert -file path/to/.crt -alias <cert alias>
-keystore <JRE/JAVA_HOME>/jre/lib/security/cacerts -storepass
changeit
```



自签名的证书仅可以用于开发和测试目的。在生产环境中使用这些证书不能提供所需的安全。在生产环境中，要始终使用由受信任的颁发机构出具并签署的证书。安全地存储你的私钥。

现在，将生成的 keystore.jks 放入 OTRS 项目的 src/main/resources 目录，以及 application.yml 后，我们可以在边缘服务器的 application.yml 中更新此信息：

```
server:
  ssl:
    key-store: classpath:keystore.jks
    key-store-password: password
    key-password: password
  port: 8765
```

重建 Zuul 服务器 JAR 以使用 HTTPS。

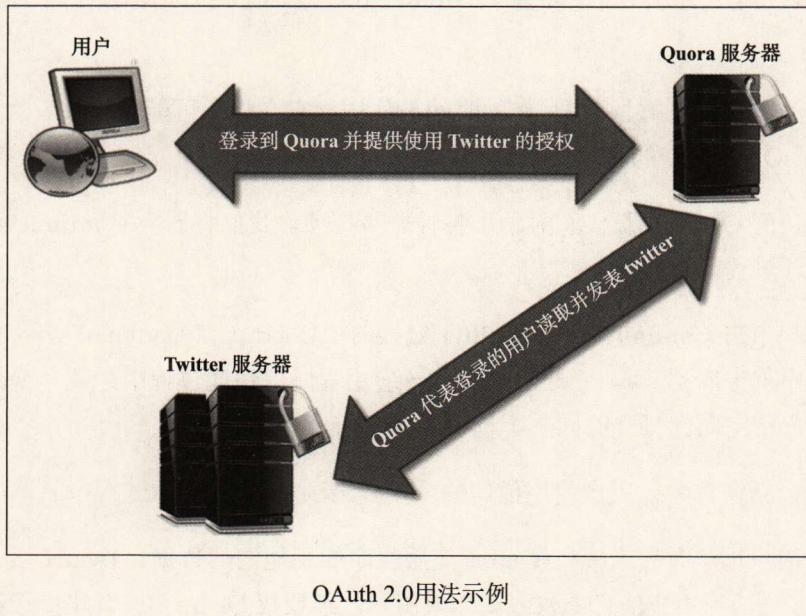


在 Tomcat 7.0.66 以上和 8.0.28 以上版本中，密钥存储文件可以存储在前面的类路径中。对于较旧的版本，可以使用密钥存储文件的路径作为 server:ssl:key-store 值。

同样，也可以为其他微服务配置 SSL。

身份验证和授权

身份验证和授权事实上是对 web 应用程序提供的。在本节中，我们将讨论身份验证和授权。在过去几年中的新范例是 OAuth。我们将学习和使用 OAuth 2.0 来实现它。OAuth 是一种开放的授权机制，它在每个主要的 web 应用程序中都实现了。Web 应用程序可以通过实现 OAuth 标准访问彼此的数据，它已成为各种 web 应用程序来验证自己的最流行的方式。比如，在 www.quora.com 上，可以使用谷歌或 Twitter 的登录 ID 注册和登录。它也是用户友好的，因为客户端应用程序（例如 www.quora.com）不需要存储用户的密码。最终用户不需要多记一个用户 ID 和密码。



OAuth 2.0

互联网工程任务组 (Internet Engineering Task Force, IETF) 规定了 OAuth 的标准和规格。在 OAuth 2.0 之前，OAuth 1.0a 曾经是最新的版本，它修复了 OAuth 1.0 中的会话固定安全缺陷。OAuth 1.0 和 1.0a 与 OAuth 2.0 有很大的不同。OAuth 1.0 依赖于安全证书和通道绑定。OAuth 2.0 不支持安全认证和通道绑定，它完全工作在传输安全层 (Transport Security Layer, TSL) 上。因此，OAuth 2.0 不提供向后兼容性。

OAuth 的用法

- 如已讨论的，它可用于身份验证。你可能已经在各种应用中看到过它，如显示使用 Facebook 登录或使用 Twitter 登录的消息。
- 应用程序可以使用它从其他应用程序读取数据，如通过把一个 Facebook 构件集成到应用程序中，或在你的博客上有一个 Twitter 的回馈。
- 或者与前一点相反的过程：允许其他应用程序访问最终用户的数据。

OAuth 2.0 规范——简明详细信息

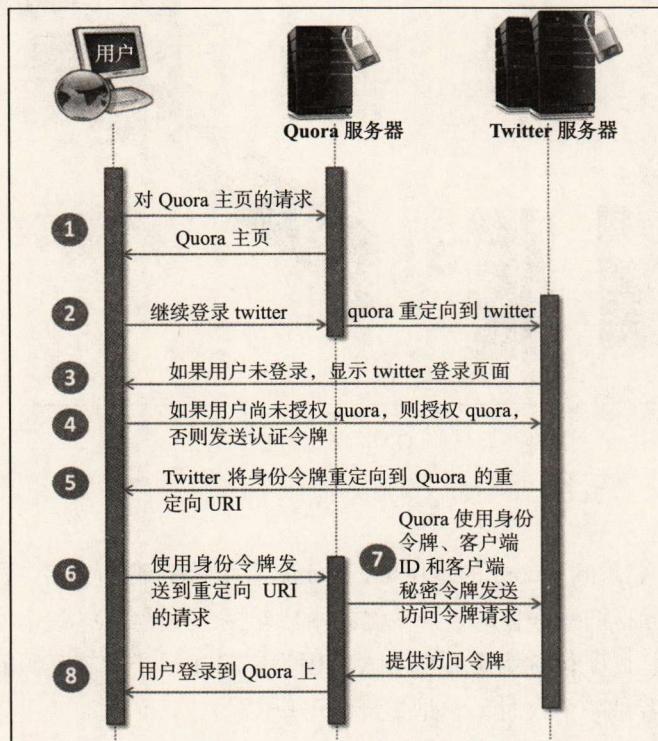
我们尝试以简明的方式讨论和理解 OAuth 2.0 规范。让我们先看看使用 Twitter 登录是如何工作的。

请注意，这里提到的过程在写作的时间被使用，它在将来可能会更改。然而，这一过程正确描述了 OAuth 2.0 的其中一个过程：

1. 用户访问 Quora 主页。它显示了各种登录选项。我们将研究 **Continue with Twitter** 的处理过程。
2. 当用户点击 **Continue with Twitter** 链接时，Quora（在 Chrome）打开一个新窗口，把用户重定向到 www.twitter.com 应用程序。在此过程中，几个 web 应用程序将用户重定向到打开的同一个选项卡/窗口上。
3. 在这个新窗口中，用户使用他们的凭据登录到 www.twitter.com。
4. 如果用户早些时候未授权 Quora 应用程序使用他们的数据，Twitter 要求用户授权 Quora 来访问用户信息的权限。如果用户已经授权 Quora，那么此步骤被跳过。
5. 经过正确的身份验证，Twitter 将用户和一个身份验证代码重定向到 Quora 的重定向 URI。
6. Quora 重定向 URI 在浏览器中输入时，Quora 发送客户端 ID、客户端秘密令牌和身份验证代码（Twitter 在步骤 5 中发送）到 Twitter。
7. Twitter 验证这些参数后，将访问令牌发送到 Quora。

8. 用户成功获取访问令牌后被登录到 Quora 上。
9. Quora 可能使用此访问令牌从 Quora 中获取用户信息。

你一定想知道 Twitter 是如何得到 Quora 的重定向 URI、客户端 ID 和秘密令牌的。Quora 充当一个客户端应用程序，而 Twitter 充当一台授权服务器。Quora 作为客户端，利用 Twitter 的 OAuth 实现，使用资源所有者（最终用户）的信息在 Twitter 上注册。Quora 在注册时提供一个重定向 URI，Twitter 把客户端 ID 和秘密令牌提供给 Quora。它以这种方式工作。在 OAuth 2.0 中，用户信息被称为用户资源。Twitter 提供一台资源服务器和授权服务器。我们将在下一节对这些 OAuth 术语进行更多的讨论。

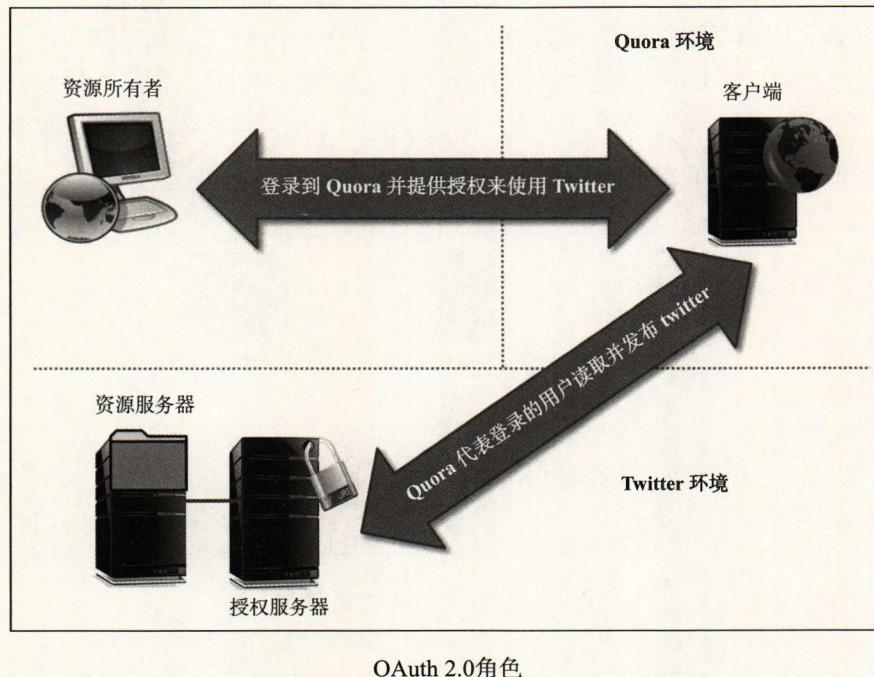


使用Twitter登录OAuth 2.0的示例过程

OAuth 2.0 角色

在 OAuth 2.0 规范中定义有 4 个角色：

- 资源所有者
- 资源服务器
- 客户端
- 授权服务器



资源所有者

在使用 Twitter 登录到 Quora 的示例中，Twitter 用户是资源所有者。资源所有者是一个实体，拥有准备要分享的受保护的资源（例如用户句柄、微博等）。这个实体可以是应用程序或一个人。我们把此实体称为资源所有者，因为它只能授予对其资源的访问。规范还定义了，当资源所有者是一个人时，它是指最终用户。

资源服务器

资源服务器承载受保护的资源。它应该有能力使用访问令牌为访问这些资源的请求提供服务。对于使用 Twitter 登录到 Quora 的示例，Twitter 是资源服务器。

客户端

在使用 Twitter 登录到 Quora 的示例中，Quora 是客户端。客户端是代表资源所有者对资源服务器发出访问受保护资源请求的应用程序。

授权服务器

授权服务器为客户端应用程序提供不同的令牌，如访问令牌或刷新令牌，令牌只有在资源所有者对自己进行身份验证后才提供。

OAuth 2.0 不为资源服务器和授权服务器之间的交互提供任何规范。因此，授权服务器既可以与资源服务器在同一服务器上，也可以是一个单独的服务器。

单个授权服务器也可以用于为多个资源服务器颁发访问令牌。

OAuth 2.0 客户端注册

与授权服务器通信的客户端首先应注册到授权服务器才能获取资源的访问键。OAuth 2.0 规范没有指定客户端注册到授权服务器的方式。注册不需要客户端和授权服务器之间的直接通信，可以使用自己发出或第三方发出断言完成注册。授权服务器使用这些断言之一来获取所需的客户端属性。让我们看看客户端属性的内容：

- 客户端类型（在下一节中讨论）。
- 客户端重定向 URI，如我们在使用 Twitter 登录到 Quora 的示例中讨论的。这是一个用于 OAuth 2.0 的端点。我们将在端点部分讨论其他端点。
- 任何授权服务器所需的其他信息，例如客户名称、描述、徽标图像、联系方式及接受法律的条款和条件等。

客户端类型

基于它们对客户端凭据的保密能力，规范中描述了两种类型的客户端：保密和公开。客户端凭据是授权服务器为了与客户端交流而向它们颁发的秘密令牌。

保密客户端类型

这种客户端应用程序安全地保持密码和其他凭据或保密地维护它们。在使用 Twitter 登录到 Quora 的示例中，Quora 的应用程序服务器是安全的，并实现了限制的访问。因此，

它是保密性质的客户类型。只有 Quora 应用程序管理员拥有对客户端凭据访问的权限。

公开客户端类型

这种客户端应用程序不 (*not*) 安全地保持密码和其他凭据或保密地维护它们。移动设备或桌面电脑上的任何本机应用程序，或在浏览器上运行的应用程序是公共客户端类型的完美例子，因为这些应用程序在其中内嵌保持客户端凭据。黑客可以破解这些应用程序，客户端凭据可能会泄露。

客户端可以是一个分布式的基于组件的应用程序，例如，它可以同时有一个 web 浏览器组件和服务器端组件。在这种情况下，这两个组件会有不同的客户端类型和安全上下文。如果授权服务器不支持此类客户端，这些客户端应该把每个组件作为单独的客户端注册。

基于 OAuth 2.0 客户端类型，客户端可以有以下配置文件：

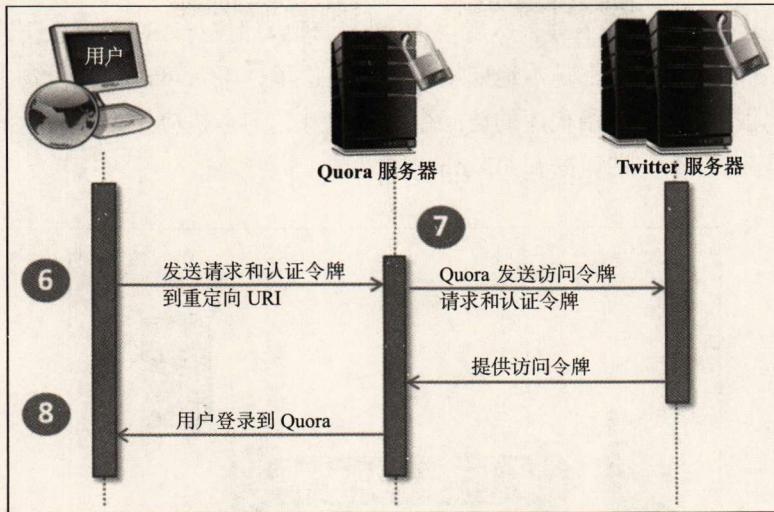
- Web 应用程序
- 基于用户代理的应用程序
- 本机应用程序

Web 应用程序

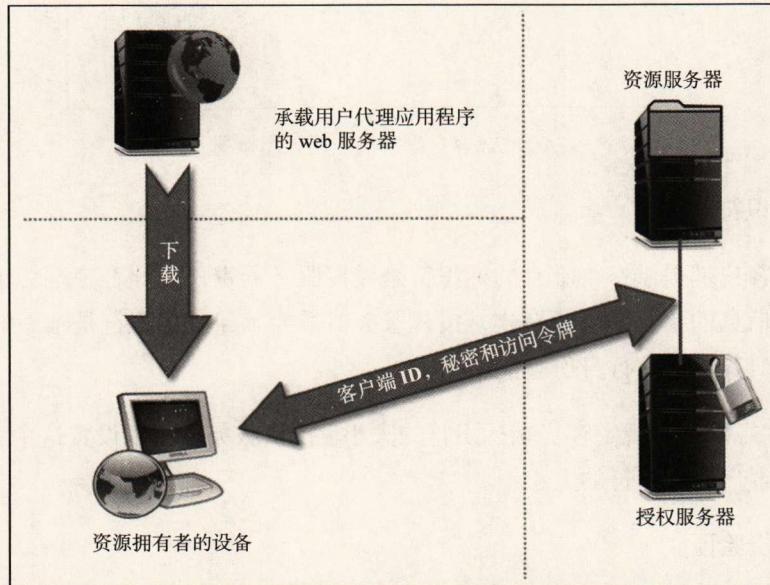
在使用 Twitter 登录到 Quora 的示例中，使用的 Quora web 应用程序是 OAuth 2.0 web 应用程序客户端配置文件的一个完美例子。Quora 是运行在 web 服务器上的保密客户端。资源所有者（最终用户）在他的设备（桌面电脑/平板电脑/手机）浏览器（用户代理）上使用 HTML 用户界面访问 Quora 上的应用程序（OAuth 2.0 客户端）。资源所有者不能访问客户端（Quora OAuth 2.0 客户端）的凭据和访问令牌，因为这些都存储在 web 服务器上。可以在 OAuth 2.0 示例流程图中看到这种行为。请参阅下页图所示的步骤 6 到 8。

基于用户代理的应用程序

基于用户代理的应用程序是公开客户端类型。在这里，虽然应用程序驻留在 web 服务器中，但资源所有者在用户代理（例如，web 浏览器）上下载它，然后再执行应用程序。在这里，驻留在资源所有者设备上的用户代理中下载的应用程序与授权服务器进行通信，资源所有者可以访问客户端凭据和访问令牌。游戏应用程序是这种应用程序配置文件的一个好例子。



OAuth 2.0客户端web应用程序配置文件

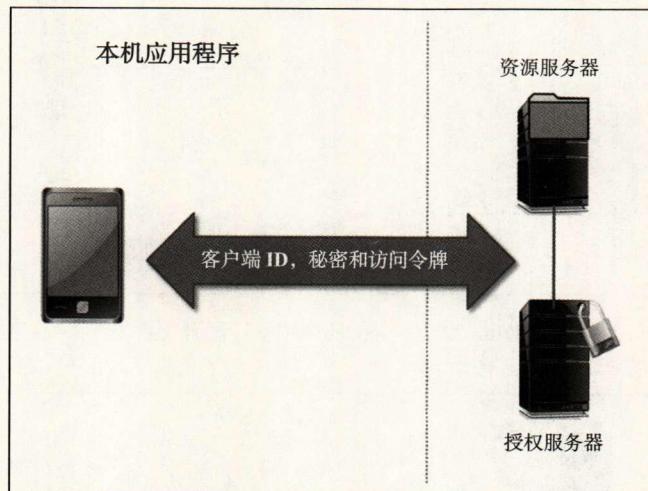


OAuth 2.0客户端用户代理应用程序概况

本机应用程序

本机应用程序类似基于用户代理的应用程序，但它们是在资源所有者的设备上安装和

执行，而不是从 web 服务器下载，然后再在用户代理内部执行。在手机上下载的许多本地客户端（移动 app）的类型都是本地应用程序。在这里，平台可确保在设备上的其他应用程序不能访问凭据和其他应用程序的访问令牌。此外，与本机应用程序通信的服务器不应与本机应用程序共享客户端凭据和 OAuth 令牌。



OAuth 2.0客户端本机应用程序概况

客户端标识符

向注册的客户端提供一个唯一的标识符是授权服务器责任。此客户端标识符是由注册的客户端提供的信息的字符串表示形式，授权服务器需要确保此标识符是唯一的。授权服务器不应使用它对其自身进行身份验证。

OAuth 2.0 规范没有规定客户端标识符的大小。授权服务器可以设置这个大小，并应该记录它颁发的客户端标识符的大小。

客户端身份验证

授权服务器应基于其客户端类型验证客户端，应该确定适合并且符合安全要求的身份验证方法。在每个请求中，它应该只使用一种身份验证方法。

通常情况下，授权服务器使用一组客户端凭据，如客户端密码和一些密钥令牌来验证保密客户端的身份。

授权服务器可以与公共客户端建立一种客户端身份验证方法。然而，出于安全原因，它不得依赖此身份验证方法来确定客户端。

拥有客户端密码的客户端可以使用 HTTP 基本身份验证。OAuth 2.0 不建议在请求正文 中发送客户端凭据，它建议在身份验证所需的端点上使用 TLS 和蛮力攻击保护。

OAuth 2.0 协议端点

端点只不过是我们使用 REST 或 web 组件如 Servlet 或 JSP 的一个 URI。OAuth 2.0 定义了三种类型的端点。其中两种是授权服务器端点，一种是客户端端点：

- 授权端点（授权服务器端点）
- 令牌端点（授权服务器端点）
- 重定向端点（客户端端点）

授权端点

此端点负责验证资源所有者的身份，并且一旦验证通过，获得授权许可（authorization grant）。我们将在下一节讨论授予许可。

授权服务器要求对授权端点采用 TLS，端点 URI 中不得包含片段组件，授权的端点必须支持 HTTP GET 方法。

规范没有规定以下内容：

- 授权服务器对客户端进行身份验证的方式。
- 客户端将如何接受授权端点的 URI。通常情况下，文档中包含授权端点的 URI，或者客户端在注册时获取它。

令牌端点

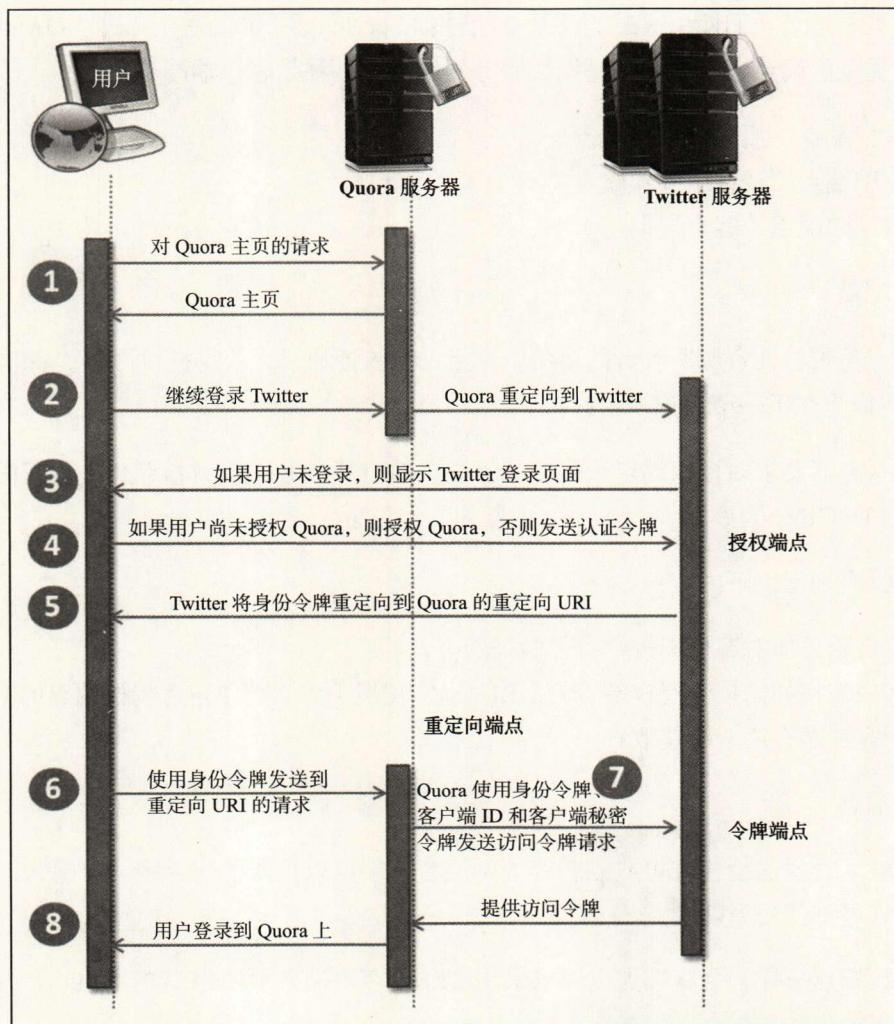
客户端通过发送授权许可或刷新令牌调用令牌端点以接受访问令牌。除隐式许可外的所有授权许可都使用令牌端点。

像授权端点一样，令牌端点也要求采用 TLS。客户端必须使用 HTTP POST 方法向令牌端点发出请求。

像授权端点一样，规范也没有指定客户端如何接收令牌端点的 URI。

重定向端点

一旦资源所有者和授权服务器之间的授权端点的交互完成，授权服务器就使用重定向端点把资源所有者的用户代理（例如 web 浏览器）重定向回客户端。客户端在注册时提供重定向端点，重定向端点必须是绝对 URI 并且不包含片段组件。



OAuth 2.0 授权类型

客户端基于从资源所有者获得的授权，向授权服务器请求访问令牌。资源所有者以授权许可的形式给出授权。OAuth 2.0 定义了四种类型的授权许可：

- 授权代码许可
- 隐式许可
- 资源所有者密码凭据许可
- 客户端凭据许可

OAuth 2.0 还提供了一种扩展机制来定义额外的许可类型。可以在官方的 OAuth 2.0 规范中研究这个。

授权代码许可

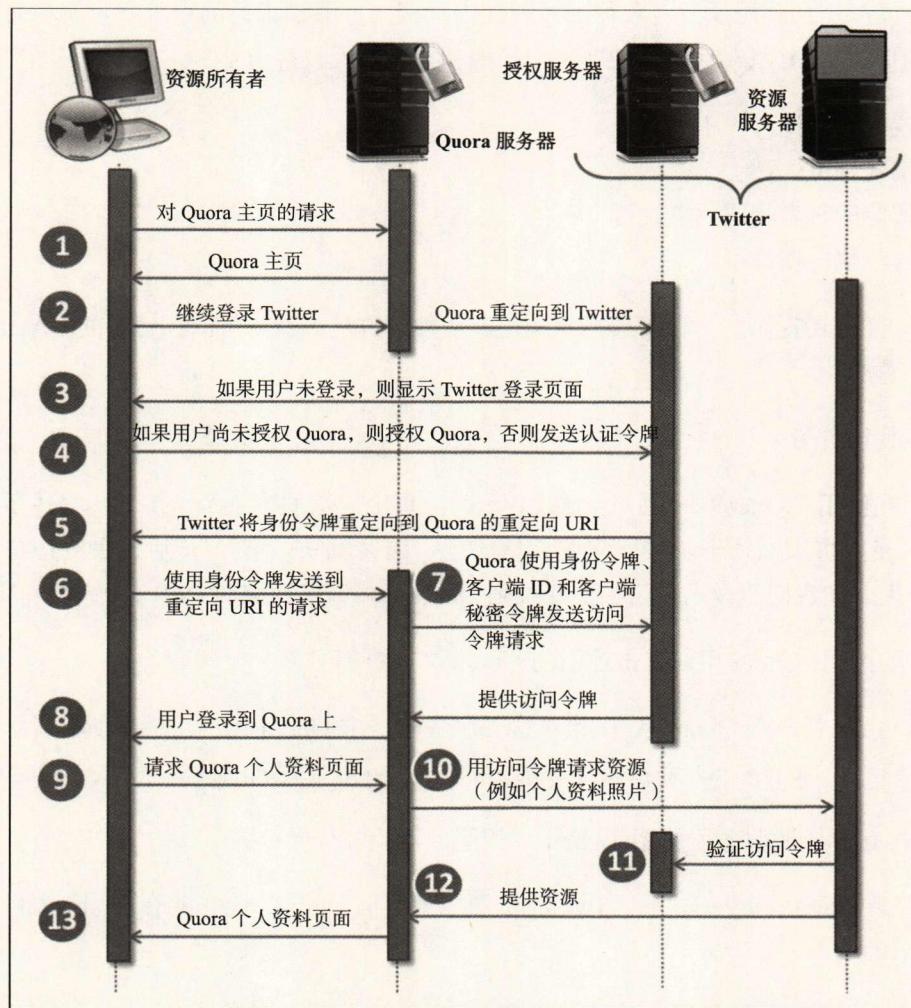
我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如 Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供了所要求的资源。
5. Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

授权代码请求和响应

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。



OAuth 2.0授权代码许可流程

对授权端点 URI 的授权请求（步骤 4）：

参 数	必需/可选	描 述
response_type	必需	code（必须使用此值）。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。
redirect_uri	可选	它表示注册时由客户端给出的重定向 URI。

续表

参数	必需/可选	描述
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。
state	推荐	客户端使用此参数来保持客户端请求和（从授权服务器的）回调之间的状态。规范建议使用它来防止跨站点请求伪造攻击。

授权响应（步骤 5）：

参数	必需/可选	描述
code	必需	授权服务器生成的代码（授权码）。 代码在它生成后应该过期，建议的最大生存期是 10 分钟。 客户端使用此代码不得超过一次。 如果客户端使用它不止一次，那么请求必须被拒绝，并且以前基于此代码发出的所有令牌都应被撤销。 代码被绑定到客户端 ID 和重定向 URI。
state	必需	它表示注册时授权服务器发给客户端的 ID。

对令牌端点 URI 的令牌请求（步骤 7）：

参数	必需/可选	描述
grant_type	必需	authorization_code（必须使用此值）。
code	必需	从授权服务器接收到的代码（授权码）。
redirect_uri	必需	如果它被列入授权代码请求中，则此参数必需，并且值应该匹配。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。

令牌响应（步骤 8）：

参数	必需/可选	描述
access_token	必需	由授权服务器颁发的访问令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。

续表

参数	必需/可选	描述
scope	可选/必需	如果客户端请求的范围是相同的，则此参数是可选的。 如果访问令牌范围不同于客户端在请求中提供的那个，则此参数必需，通知客户端授予的访问令牌的实际范围。 如果客户端请求访问令牌时未提供范围，那么授权服务器应提供默认范围，或拒绝该请求，指示无效的范围。

错误响应：

参数	必需/可选	描述
error	必需	规范中定义的一个错误代码，例如， unauthorized_client、invalid_scope。
error_description	可选	错误的简短说明。
error_uri	可选	描述错误的错误页的 URI。

如果客户端在授权请求中传递了状态，则附加错误参数状态也在错误响应中发送。

隐式许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

隐式许可请求和响应

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权

服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对授权端点 URI 的授权请求：

参 数	必需/可选	描 述
response_type	必需	Token (必须使用此值)。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。
redirect_uri	可选	它表示注册时由客户端给出的重定向 URI。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。
state	推荐	客户端使用此参数来保持客户端请求和(从授权服务器的)回调之间的状态。规范建议使用它来防止跨站点请求伪造攻击。

访问令牌的响应：

参 数	必需/可选	描 述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。
scope	可选/必需	如果客户端请求的范围是相同的，则此参数是可选的。 如果访问令牌范围不同于客户端在请求中提供的那个，则此参数必需，通知客户端授予的访问令牌的实际范围。 如果客户端请求访问令牌时未提供范围，那么授权服务器应提供默认范围，或拒绝该请求，指示无效的范围。
state	可选/必需	如果状态是在客户端授权请求中传入的，则此参数必需。

错误响应：

参 数	必需/可选	描 述
error	必需	规范中定义的一个错误代码，例如，unauthorized_client、invalid_scope。

续表

参数	必需/可选	描述
error_description	可选	错误的简短说明。
error_uri	可选	描述错误的错误页的 URI。

如果客户端在授权请求中传递了状态，则附加错误参数状态也在错误响应中发送。

资源所有者密码凭据许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可。我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

资源所有者密码凭据许可请求和响应。

正如你在上一节中看到的，在授权代码流程的所有步骤（共 13 步）中，可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对令牌端点 URI 的访问令牌请求：

参数	必需/可选	描述
grant_type	必需	password（必须使用此值）。
username	必需	资源所有者的用户名。

续表

参 数	必需/可选	描 述
password	必需	资源所有者密码。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。

访问令牌响应（步骤 8）：

参 数	必需/可选	描 述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。
可选参数	可选	额外的参数。

客户端凭据许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

客户端凭据许可请求和响应。

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对令牌端点 URI 的访问令牌请求：

参 数	必需/可选	描 述
grant_type	必需	client_credentials (必须使用此值)。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。

访问令牌响应：

参数	必需/可选	描述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。

使用 Spring Security 的 OAuth 实现

OAuth 2.0 是保护 API 的一种方式。Spring Security 提供了 Spring Cloud Security 和 Spring Cloud OAuth2 组件来实现我们上面讨论的许可流程。

我们会再多创建一个服务，安全服务（security-service），它将控制身份验证和授权。

创建一个新的 Maven 项目，并按照下列步骤操作：

1. 在 pom.xml 中添加 Spring Security 和 Spring Security OAuth2 依赖项：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

2. 在应用程序类中使用@EnableResourceServer注解。这将允许此应用程序可以作为资源服务器工作。根据 OAuth 2.0 规格，@EnableAuthorizationServer是我们启用授权服务器要使用的另一个注解：

```
@SpringBootApplication
@RestController
@EnableResourceServer
public class SecurityApp {

    @RequestMapping("/user")
    public Principal user(Principal user) {
        return user;
    }

    public static void main(String[] args) {
        SpringApplication.run(SecurityApp.class, args);
    }

    @Configuration
    @EnableAuthorizationServer
    protected static class OAuth2Config extends
    AuthorizationServerConfigurerAdapter {

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer
endpointsConfigurer) throws Exception {
            endpointsConfigurer.authenticationManager(authenticationManager);
        }
    }
}
```

```
    @Override
    public void configure(ClientDetailsServiceConfigurer
clientDetailsServiceConfigurer) throws Exception {
    // 使用硬编码的内存中机制，因为它只是一个示例

        clientDetailsServiceConfigurer.inMemory()
            .withClient("acme")
            .secret("acmesecret")
            .authorizedGrantTypes("authorization_code", "refresh_
token", "implicit", "password", "client_credentials")
            .scopes("webshop");
    }
}
}
```

3. 在 application.yml 中更新安全服务配置，如下面的代码所示：

- server.contextPath: 它表示上下文路径。
- security.user.password: 我们会为本演示使用硬编码的密码。在实际使用中可以重新配置它：

```
application.yml
info:
    component:
        Security Server

server:
    port: 9001
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    contextPath: /auth

security:
    user:
        password: password
```

```
logging:  
  level:  
    org.springframework.security: DEBUG
```

现在我们有了安全服务器，我们会使用新的微服务 api-service 公开我们的 API，这将用于与外部应用程序和 UI 进行通信。

创建一个新的 Maven 项目，并按照下列步骤操作：

1. 在 pom.xml 中添加 Spring Security 和 Spring Security OAuth2 依赖项：

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-undertow</artifactId>  
</dependency>  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
  
<dependency>  
  <groupId>com.packtpub.mmj</groupId>  
  <artifactId>online-table-reservation-common</artifactId>  
  <version>PACKT-SNAPSHOT</version>  
</dependency>  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-security</artifactId>  
</dependency>  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-oauth2</artifactId>  
</dependency>  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-eureka</artifactId>  
</dependency>  
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <!-- Testing starter -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

2. 在应用程序类中使用@EnableResourceServer注解。这将允许此应用程序作为资源服务器：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
@EnableResourceServer
@ComponentScan({"com.packtpub.mmj.api.service", "com.packtpub.mmj.common"})
public class ApiApp {

    private static final Logger LOG = LoggerFactory.
```