



下载APP



## 12 | 实战：将 ToDo 应用扩展为一个 REST 服务

2021-08-30 郑晔

《程序员的测试课》

课程介绍 &gt;




讲述：郑晔

时长 17:28 大小 16.01M



你好，我是郑晔！

经过了基础篇的介绍，相信你已经对在日常开发中测试应该做到什么程度有了一个初步的认识。有了基础固然好，但对于很多人来说，面对常见的场景还是不知道如何下手。在接下来的应用篇中，我们就用一些开发中常见的场景，给你具体介绍一下怎么样把我们学到的知识应用起来。

在后端开发中，最常见的一种情况就是开发一个 REST 服务，将数据写到数据库里面，<sup>++</sup>就是传说中的 CRUD 操作。这一讲，我们就把前面已经写好的 [ToDo 应用](#) 扩展一下  它变成一个 REST 服务。

### 扩展前的准备

具体动手写任何代码之前，我们先要搞清楚我们要把这个应用改造成什么样子。把 ToDo 应用扩展为一个 REST 服务也就是说，原来本地的操作现在要以 REST 服务的方式提供了。另外，在这次改造里面，我们还会把原来基于文件的 Repository 改写成基于数据库的 Repository，这样，就和大多数人在实际的项目中遇到的情况是类似的了。

有人可能会想，既然是 REST 服务，那是不是要考虑多用户之类的场景。你可以暂时把它理解成一个本地运行的服务（也就是说只有你一个人在使用），所以我们可以不考虑多用户的情况。这样做可以让我们把注意力更多放在测试本身上，而增加更多的能力是需求实现的事情，你可以在后面拿这个项目练手时，做更多的尝试。

确定好了需求目标，接下来，我们就要进入到具体的实现过程里面了。RESTful API 不同于命令行应用，不应该把它的代码同命令行的代码混杂在一起，所以，我们可以建一个单独的模块来放置这些代码，我把这个模块叫 todo-api。至于具体采用的技术栈，我们就使用在 Java 社区最常用的 Spring Boot，Spring Boot 能够极大简化了 REST 服务的开发。

同之前一样，我们先实现 Repository 的部分，然后再来做接口。或许你会有一个疑问，难道不是要实现业务核心部分吗？别忘了，我们在**之前的实现中特意将业务核心部分隔离了出来，让它不依赖于任何具体的外部实现**。虽然我们是将一个命令行应用改成一个 RESTful API，但业务核心部分并没有发生任何改变，所以，我们也不需要重新编写一份。这就是软件设计的价值所在。

## 数据访问

前面说过，我们要把之前基于文件版本的 Repository 实现改成基于数据库的版本，所以我们要先来确定数据访问相关的技术。我选择 MySQL 这个大家最常用的数据库，访问数据库的程序库我选择的是 Spring Data JPA，因为它可以让我尽可能少编写代码。

## 技术选型

两种常见的访问数据库的方式分别是 MyBatis 和 JPA。MyBatis 倾向于让人手工编写 SQL 语句，而 JPA 则采用更加面向对象的角度，它访问数据库的 SQL 语句常常是由框架生成的。二者的差异主要是 MyBatis 更加面向具体的实现，而 JPA 则提供了更好的抽象能力。

目前国内的现状是很多团队会使用 MyBatis，他们给出的理由大多是自己写 SQL 比较好控制，尤其是对一些复杂场景来说更容易优化。不过，实际情况往往是，如果采用 JPA 的


话，很多团队对于生成什么样的代码自己完全心里没有数，因为欠缺建模能力才用 MyBatis。而对于很多建模做得比较好的团队来说，使用 JPA 往往开发效率更高。

Spring Data JPA 在 JPA 上提供了进一步的封装，一些常见的数据访问甚至不需要去编写代码，因为访问数据库的 SQL 都是由框架生成的，是一个标准操作。因为不是我们编写的代码，我们也无需验证它的正确性，只要保证我们自己写的代码正确地表达了我们的意图即可。如果真的有一些比较复杂的 SQL 逻辑要实现，Spring Data JPA 也允许我们自己手写 SQL，这是框架留给我们的优化手段。

所以，我们这里选择 Spring Data JPA。下面我们就来开始我们的实现之旅。


## 数据库迁移

在开始编码测试工作之前，我们要先确定 Todo 项存储的结构。所以，我们要在数据库中创建一个表。

 复制代码

```
1 CREATE TABLE todo_items (  
2     `id` int auto_increment,  
3     `content` varchar(255) not null,  
4     `done` tinyint not null default 0,  
5     primary key (`id`)  
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

我们已经在实战中看见过实体的样子了，所以，这里的表结构并不难理解。唯一需要稍微解释一下的就是在表里面我们用了 id，而在 Todo 项的实体中，它对应的是 index。其实，只要你稍微仔细地想一下就不难发现，在我们之前的设计中，index 就是起到了 id 的作用。对应的实体就是下面这样：

 复制代码

```
1 @Entity  
2 @Table(name = "todo_items")  
3 public class TodoItem {  
4     @Id  
5     @Column(name = "id")  
6     @GeneratedValue(strategy = GenerationType.IDENTITY)  
7     private long index;  
8  
9     @Column
```

```
10     private String content;
11
12     @Column
13     private boolean done;
14     ...
15 }
```

在项目自动化中，数据库迁移脚本我们采用了 [Flyway](#)，它可以很方便地将数据库的变更管理起来。我们只要在 `$rootDir/gradle/config/migration` 这个位置创建一个迁移脚本，把上面的 SQL 写进去就好，具体的细节你可以参考我们的开源项目。

有了迁移脚本，我们就可以执行命令将这个表创建出来。

```
1 ./gradlew flywayMigrate
```

[复制代码](#)

好，基础已经准备好了，我们准备要动手写测试了。

## 编写测试

我在上一讲说过，测试数据库相关的内容属于兼具集成测试和单元测试两种属性的测试，一方面，它要对数据库做集成，另一方面，它要测的内容本身属于验证一个单元代码是否编写正确的范畴。对于数据库相关的测试，Spring 提供了很好的支持，让我们可以更好地完成验证工作。

下面就是一个测试。如果你还记得之前文件版本 Repository 的测试，这个测试你可能会很眼熟。没错，这里的测试我几乎就是原封不动地把前面的测试搬了过来，因为 Repository 接口的行为几乎是一致的。这也是我这里并没有做测试场景分析的原因。

```
1 @ExtendWith(SpringExtension.class)
2 @DataJpaTest
3 @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
4 @TestPropertySource("classpath:test.properties")
5 public class TodoItemRepositoryTest {
6     @Autowired
7     private TodoItemRepository repository;
8
9     @Test
```

[复制代码](#)

```
10     public void should_find_nothing_for_empty_repository() {
11         final Iterable<TodoItem> items = repository.findAll();
12         assertThat(items).hasSize(0);
13     }
14     ...
15 }
```

看到 `@Autowired`，如果你熟悉 Spring 应该感到非常亲切，它表示这个字段由框架自动绑定的。那这个自动绑定为什么能起作用呢？这就要拜前面几个 Annotation 所赐了。

`@ExtendWith(SpringExtension.class)`，在这里面，`@ExtendWith` 是 JUnit 5 提供的扩展机制，让第三方有机会编写自己的代码。而 `SpringExtension` 就是 Spring 提供的扩展，用来做一些 Spring 自己需要的准备和清理之类的工作，比如依赖注入就是通过它完成的。

`@DataJpaTest`，表示这个测试采用 Spring Data JPA。有了这个 Annotation，Spring 框架会替我们把 `Repository` 的实例生成出来。因为使用 Spring Data JPA 的时候，我们只编写了接口。还记得 `TodoItemRepository` 这个接口吗？现在它变成了下面这个样子。

[复制代码](#)


```
1 public interface TodoItemRepository extends Repository<TodoItem, Long> {
2     TodoItem save(TodoItem item);
3
4     Iterable<TodoItem> findAll();
5 }
```

同之前相比，这里的方法没用任何变化，只是扩展了一个接口 `Repository`，这是一个标记接口，也就是意味着只有接口，没有方法。实现这个接口是 Spring Data JPA 的要求，它会在运行时为这个接口生成相应的实例，换言之，我们不需要为此编写具体的实现。

其实，Spring Data JPA 的函数名是有一些约定的，在前面给 `Repository` 的函数命名的时候，我就是参考了 Spring Data JPA 的命名规则，所以，我们在这里可以无缝地与 Spring Data JPA 对接在一起。

按照 Spring Data JPA 的要求，我们要让 Spring 在启动的时候能够找到我们配置的实体和 `Repository`。因为我们这里的实体和 `Repository` 不在缺省的扫描路径上，所以这里需要单独配置一下。下面就是我们的配置，这是一个典型的 Spring Boot 的应用。



 复制代码

```
1 @SpringBootApplication
2 @EnableJpaRepositories({"com.github.dreamhead.todo.core"})
3 @EntityScan({"com.github.dreamhead.todo.core"})
4 public class Bootstrap {
5     ...
6 }
```

万事俱备，我们现在可以运行测试了，如果一切顺利的话，测试会一次性运行通过。

这里其实有个实现的细节，测试并没有在数据库留下任何痕迹，正如我们在讲集成测试中说过的那样，这里的测试在运行之后回滚了在测试过程中插入的数据，这是 DataJpaTest 的缺省行为，大大简化了测试的难度。

你会发现，其实我们并没有写多少有逻辑的代码：表是 SQL 语句生成的，测试是从前面的测试搬过来的，主要的工作都是配置，而数据库访问的过程是框架生成的。减少自己编码的工作量，我们的测试压力也就小了很多。

## RESTful API

有了 Repository，接下来，我们就要来设计实现 API 接口了。对于一个服务而言，对外提供哪些接口是很重要的。任何一个提供后端服务的团队都要仔细地设计其服务接口，确定它应该提供哪些能力，而不仅仅是围绕着前端需求去做。

## 设计 RESTful API

还记得我们的 ToDo 应用提供了哪些能力吗？我们回顾一下：

添加一个 Todo 项；

完成一个 Todo 项；

Todo 项列表。

接下来，我们就把它们设计成 API 接口。所有这三个能力都是围绕着 Todo 项进行的，所以，我们可以把它们设计在一个资源下，不妨就把它的 URI 设计成 /todo-items，一般来说，这里一般会使用复数，表示这有一堆资源。

有了最基础的资源，接下来，就是一个一个地按照 RESTful API 的方式设计出来。**首先是添加一个 Todo 项。**按照通常 RESTful API 接口的设计方式，相当于在服务端创建了一个新的资源，而创建的语义一般会用 POST 请求表示。创建一个 Todo 项，主要包含的就是 Todo 项的内容，其格式我们就采用 RESTful API 常用的 JSON 格式了。

[复制代码](#)

```
1 POST /todo-items
2
3 {
4   "content": "foo"
5 }
```

**有了创建 Todo 项的服务，接下来就是完成一个 Todo 项了。**完成一个 Todo 项，按照 RESTful API 的设计方式，这个动作相当于对已有资源的修改，修改对应的 HTTP 动词是 PUT。不同于 POST，PUT 操作需要指定一个具体的资源，我们这里使用索引作为唯一标识，其对应的内容就是完成字段（done）置为 true。目前来说，我们也不支持其它的处理，所以严格地说，这里的内容其实意义不大。

[复制代码](#)

```
1 PUT /todo-items/{index}
2
3 {
4   done: true
5 }
```

**最后是一个 Todo 项列表。**列表操作实际上是一种查询，在 RESTful API 设计中，查询对应的 HTTP 动词是 GET。在我们的实战需求中，Todo 项列表还分为查询未完成的 Todo 项和查询所有，从查询的角度来看，就是查询的参数不同。我们这里设置查询参数为 all，缺省情况下 all 的值为 false，如果显示设置了这个值，则按照设置的值进行查询。


[复制代码](#)

```
1 GET /todo-items?all=true
```

## 测试 RESTful API

做好了基本的设计工作，接下来我们就该进入代码编写的环节了。

同 Repository 部分一样，我们在这个部分的测试也准备从之前的测试中借鉴过来。所以，我们这里不把重点放在测试场景的分析上，而是来讨论如何编写测试。下面就是一个测试。

 复制代码

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 @Transactional
4 public class TodoItemResourceTest {
5     @Autowired
6     private MockMvc mockMvc;
7     @Autowired
8     private TodoItemRepository repository;
9
10    ...
11
12    .
13    @Test
14    public void should_add_item() throws Exception {
15        String todoItem = "{ " +
16            "\"content\": \"foo\"" +
17            "}";
18        mockMvc.perform(MockMvcRequestBuilders.post("/todo-items")
19            .contentType(MediaType.APPLICATION_JSON)
20            .content(todoItem))
21            .andExpect(status().isCreated());
22        assertThat(repository.findAll()).anyMatch(item -> item.getContent().eq
23    }
24
25    ...
26 }
```

从测试的名字便不难看出，这个测试是用来测试添加 Todo 项的。在这个类的开头有几个 Annotation：

@SpringBootTest，它告诉我们，接下来的测试是把所有组件都集成起来的集成测试。在前面的实战中，我说过最外面的接口很薄，所以我把集成测试和单元测试的工作量放到了一起。

@AutoConfigureMockMvc，表示我们要使用的是模拟的网络环境，也就不是真实的网络环境，这样做可以让访问速度快一些。



@Transactional，说明这个测试是事务性的，在缺省的测试事务中，执行完测试之后，数据是要回滚，也就是不对数据库造成实际的影响。这要单独标记，否则就会有数据写入到数据库里面。而之前的 @DataJpaTest 自身就包含了这个 Annotation，所以不用特别声明。

有了这些基础准备，我们就可以测试了。你可以认为，当我们执行测试时服务已经起好了，我们这里就像一个普通的客户端一样去访问一个服务，核心的部分就是下面这段代码。

[复制代码](#)

```
1 todoItem = "{ " +
2           "\"content\": \"foo\"" +
3           "}";
4 mockMvc.perform(MockMvcRequestBuilders.post("/todo-items")
5               .contentType(MediaType.APPLICATION_JSON)
6               .content(todoItem))
7       .andExpect(status().isCreated());
```

我们创建了一个请求，设置了这个请求的基本信息，用什么样的 HTTP 动词（POST）访问哪个地址（/todo-items），具体的内容是什么等等。然后，预期返回的参数是什么（状态码是 201，也就是 CREATED）。

这里我们用的是 MockMVC，因为我们配置了 @AutoConfigureMockMvc，它给我们创建了一个模拟的网络环境。这就是 Spring 在测试方面做得好的地方，作为框架的使用者，我们面对的都是编程的接口，支撑这些接口的实现在正常情况下是标准的网络环境，但 Spring 为我们提供了测试专用的实现，也就是不同的运行时，这就是做好了软件设计的结果。

不同于直接调用接口进行单元测试，这里的测试是集成测试，走的是完整的路径。所以，我们可以测试一些属于外部接口的行为，比如我们可以测试传入空的字符串该怎么办。

[复制代码](#)

```
1 @Test
2 public void should_fail_to_add_unknown_request() throws Exception {
3     String todoItem = "";
4
5     mockMvc.perform(MockMvcRequestBuilders.post("/todo-items")
6         .contentType(MediaType.APPLICATION_JSON)
```

```
7         .content(todoItem))
8         .andExpect(status().is4xxClientError());
9     }
```

## 编写 RESTful API

有了测试，接下来就是实现相应的代码了。

[复制代码](#)

```
1 @RestController
2 @RequestMapping("/todo-items")
3 public class TodoItemResource {
4     private TodoItemService service;
5
6     @Autowired
7     public TodoItemResource(final TodoItemService service) {
8         this.service = service;
9     }
10
11     @PostMapping
12     public ResponseEntity addTodoItem(@RequestBody final AddTodoItemRequest re
13         if (Strings.isNullOrEmpty(request.getContent())) {
14         return ResponseEntity.badRequest().build();
15     }
16
17     final TodoParameter parameter = TodoParameter.of(request.getContent())
18     final TodoItem todoItem = this.service.addTodoItem(parameter);
19
20     final URI uri = ServletUriComponentsBuilder
21         .fromCurrentRequest()
22         .path("/{id}")
23         .buildAndExpand(todoItem.getIndex())
24         .toUri();
25     return ResponseEntity.created(uri).build();
26 }
27 ...
28 }
```

如果你熟悉 Spring Boot 的话，这段代码对你来说应该不难。即便你不熟悉，仅仅是通过阅读代码，也很容易理解这段代码的含义：

@RestController，告诉 Spring 这是一个 REST 服务的入口类。这个类的命名是 TodoItemResource，因为在 REST 服务中，资源是一个很重要的概念，而这里的 Controller，可以说是从历史遗留的产物。

@RequestMapping( "/todo-items" ), 说明服务入口的地址是 /todo-items , 是这个类里所有服务的根。

每个具体的方法都会有自己相应的配置，对应着一个具体的服务，比如，在 addTodoItem 中是 @PostMapping，表示这个方法接收的是 POST 请求。

POST 服务一般都会有一个请求体，在这个方法中，我们使用 AddTodoItemRequest 的实例来接收这个请求体。在 HTTP 传输过程中传输的是文本，Spring 框架会替我们将文本转换成一个对象。只要我们把转换规则声明出来，Spring Boot 采用的 JSON 处理框架是 Jackson，所以，我们要在类的声明时采用 Jackson 的规则，就像下面这样。

[复制代码](#)

```
1 public class AddTodoItemRequest {
2     @Getter
3     private String content;
4
5     @JsonCreator
6     public AddTodoItemRequest(@JsonProperty("content") final String content) {
7         this.content = content;
8     }
9 }
```

在这里，@JsonCreator 表示这是一个 JSON 对象的构造方法，而 @JsonProperty 则表示将对应属性的值赋值给这里的参数。

从软件设计的角度说，Resource 是一个防腐层，AddTodoItemRequest 是一个外部请求对象。把外部对象和内部对象分开，这是很重要的（我在《代码之丑》中[分析过这种做法的原因](#)）。所以，在具体的函数中，我们首先要做就是把外部对象转换成内部对象。

[复制代码](#)

```
1 final TodoParameter parameter = TodoParameter.of(request.getContent());
```

好，到这里，我们把这段代码中主要的设计考量都已经分析过了。这段代码完整的实现，你可以参考我们的开源项目。

## 总结时刻

这一讲，我们将原本的 ToDo 应用从一个命令行应用扩展为一个 REST 服务。因为我们已经构建好了业务核心，所以这里的工作同之前是一样的：要增加一个 Repository，要编写服务的入口。

在增加 Repository 方面，我们选择了 Spring Data JPA，目的是减少代码的编写。然后我们增加了相应的数据库迁移脚本，这里采用 Flyway 管理数据库迁移的工作。

因为选择了 Spring Data JPA，我们在测试里用 @DataJpaTest，它会帮我们设置好 Repository，也会帮我们在测试运行之后回滚数据。

对外的接口我们采用 RESTful API 的设计。这里我们同样采用了集成测试代替单元测试的做法，集成测试是靠 @SpringBootTest 把各种组件都集成起来。这里我们还用到了 MockMvc 让我们的测试不依赖于真实的环境，访问速度可以稍微快一点点。


接口层本身是一个典型的防腐层，所以一般来说这层会做得非常薄，会把外部请求与业务层分隔开来。

如果今天的内容你只能记住一句话，那么请记住，**集成测试回滚数据，保证测试的可重复性。**

## 思考题

今天我们用 Spring 的基础设施演示了如何进行测试。你使用过 Spring 吗？有哪些测试特性让你印象深刻的？或者你用哪个框架给你提供了很好地测试支持呢？欢迎在留言区分享你的经验。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 集成测试：单元测试可以解决所有问题吗？

下一篇 13 | 在 Spring 项目中如何进行单元测试？

## 精选留言 (4)

[写留言](#)**不二先生**

2021-09-02

郑老师，你好：

有一个问题想请教下。

MockMVC 创建的模拟网络环境可以连接到数据库？这个数据库是本地的吗？

作者回复: MockMVC只是网络环境，与数据库无关，是否连接数据库是我们自己的事，可以连，也可以不连，在第14讲，我们可以看到具体怎么做。

**Fredo**

2021-08-31

老师你好，更新了一下ToDo项目 build无法通过了，task migrateToDev 这里是不还有漏了啥没上传的

作者回复: 需要你在本地的MySQL中创建数据库，这是第14讲的内容。

**大碗**

2021-08-31

有几个问题请教下老师：

1, 参数校验的逻辑在core层也有，算不算重复？现在写在的api里，能否移动到request的构造函数里面判断，然后抛出全局异常再返回BadRequest？实际业务一个add有好几个字段要check，写起来api的函数就好长了

2，测试接口的时候，构造request的使用的字符串json，为什么不用new对象再用工具t...  
展开

作者回复: 1. 不算，它俩校验参数是分别针对自己的目标。这就像你不应该担心第三方程序库里所做的校验一样。api里的代码处理的就是与Spring接口的部分，放到Request里面会造成过多的代码与这些框架产生耦合。

2. 可以，我这里因为它简单，就直接写了。

3. 你可以看看第14讲



**闻人**

2021-08-30

文中外部对象转为内部对象的实现有必要放到单独的类里吗，减少两个对象的依赖

作者回复: 简单地说，业务核心的代码不应该依赖于外部的请求，所以，外部请求对象不应该直接传到业务核心里面。

如果你想看更多的解释，可以去阅读《代码之丑》的第 11 讲。

