

本笔记是从范德堡大学的 [Building Cloud Services with the Java Spring Framework](#) 整理而来，文档并没有仔细校对过。仅供学习使用，如果侵犯任何个体或组织的权益，请联系我进行删除。

目录

第一单元.....	1
http 上传文件.....	1
响应码.....	2
Cookies.....	2
http 层之上的架构.....	3
轮询.....	3
REST.....	3
消息推送.....	4
课后习题.....	5
第二单元.....	7
Java 如何处理 http 请求.....	7
web.xml.....	9
注入攻击.....	10
Spring 框架中的 Dispatcher Servlet 和 Controller 抽象.....	10
使用 RequestParam 和 PathVar 注释来接受客户端数据.....	11
处理 multi-part 数据.....	12
课后习题.....	13
第三单元.....	15
使用 ResponseBody 注释来创建响应.....	15
使用 Jackson 框架的注释进行序列化(Marshalling).....	15
Spring Boot&应用结构.....	16
服务器伸缩.....	17
负载均衡.....	18
云平台的自动伸缩扩容技术（auto-scaling）.....	19
IaaS vs. PaaS.....	20
Spring 的依赖注入和自动装配.....	20
课后习题.....	24
第四单元.....	26
数据库映射.....	26
Java 持久化 API (JPA)	26
Spring 的仓库.....	27
SQL 注入攻击.....	29
Spring data 代码.....	30
会话 (Sessions)	32
Spring Data REST 代码.....	33

第一单元

http 上传文件

`multipart` 很擅长上传大量数据。它每一部分可以有一个单独的类型，多用于一次上传多种不同类型的数据。

如果只是需要上传少量的键值对，使用 `URL encoded` 会更快更有效率。

根据我们需要上传的数据的数量和类型，我们可以要么选择 `URL encoded` 或者 `multipart` 作为 `content type`。但是目前有另一种数据格式正在兴起，那就是 `application/json`。

上传数据的时候，我们如何得知服务器发生了什么？我们不能直接去看到服务器里是不是出错了还是上传成功了。所以 `http` 响应中很重要的一条信息就是 `status line`，其中 `HTTP` 状态码告诉了我们是否找到了文件，是否出了错。状态码告诉了客户端，服务器中发生了什么。第二个部分是一个文本，来详细描述了响应码的内容。

`HTTP` 响应的第二个部分是一系列的头（`headers`）。服务器可以给客户端发送元数据（`metadata`），并且告诉客户端该如何解释这个响应中数据。

`header` 的很重要的内容就是 `content-type`，也就是响应中返回的数据的格式。

`HTTP` 最终一个部分就是 `body`，其中包含了真实的数据本身，比如 `text/HTML`，或者请求图片的话，就是返回 `image/jpeg` 的二进制数据。

响应码

首先，我们介绍响应码的具体格式。

`1XX`，是 `info message`。

`2XX`，是 `success`。比如请求找一个文件，找到了。其中，`200` 是 `OK`。

`3XX`，是重定向，比如一些东西被移动了。客户端需要做一些行为来完成这个请求。比如客户端请求了一个资源，但是这个资源在服务器上已经被移动了。服务器就用这个状态码告诉客户端这件事情。需要客户端重新发送这个请求，来真正得到资源。

`4XX`，代表搞砸了，是客户端错误。比如 `404` 是找不到请求的资源。

`5XX`，是服务器错误。`500` 是服务器哪里出错了，需要告诉客户端请求没有被处理。

在我们构建客户端的时候，我们就要针对不同的响应码做出采取合适的行为。比如我们从服务器中请求一张图片，如果返回的是 `2XX`，我们就可以把返回的数据作为图片解析。但如果返回的是 `5XX`，就不应该作为图片解析。

Cookies

客户端是 `HTTP` 请求的主体。有时候服务器希望告诉客户端，它希望得到什么样的信息。或者说，在一系列的请求中，服务器需要判断这个客户端和之前发送请求的客户端是同一个客户端。因为可能客户端在登陆了以后，就去做了一些别的事情，下一次发送请求的时候，

服务器就需要客户端发送一些额外的信息来证明之前是这个客户端登陆的。换句话说，客户端需要存储一些信息在本地，来帮助未来的服务器分辨出，或者恢复这个客户端的某些状态。这种机制就是 **cookies**。**cookies** 就是一些服务器传给客户端的很小的数据信息，并且服务器要求客户端记住。因为客户端可能是手机，也有可能是超级计算机。所以我们不希望 **cookies** 占据了很大的空间，所以 **cookies** 只是非常非常有限的一些数据。客户端会把 **cookies** 存在临时的空间里，在将来请求的时候一起发送回服务器。这样服务器就能够分辨出这个客户端之前是否已经登陆过了。**cookies** 机制对于整个会话（**session**）来说非常重要。服务器在发送 **cookie** 给客户端的时候会设置其消亡时间，以及其信息敏感程度。客户端不会超过消亡时间的 **cookie**，因为服务器认为这已经不是一个有效的登陆了。

目前我们所提到的 **HTTP** 都是传输未加密的数据，等一下我们会讨论 **HTTPS** 传输加密的数据。如果我们想保护 **cookie** 的安全，我们可以设定客户端当且仅当使用 **HTTPS** 通讯协议的时候，才向服务器发送 **cookie**。

http 层之上的架构

HTTP 协议层是在 **TCP/IP** 层之上的。在 **HTTP** 层之上，可能有一些我们创建的层。比如说和我们视频服务交互的层。一种在 **HTTP** 层之上的是网络服务（**web services**），我们使用网络服务定义语言（**WSDL**）或者使用简单对象访问协议（**SOAP**）来发送信息。使用 **WSDL** 或者 **SOAP** 只是意味你有一个服务是通过 **HTTP** 来访问的。最近 **REST** 非常出名，**REST**（表述性状态转移）是一个非常具体的规则集合来建构 **HTTP** 层之上的协议和接口。我们说一个 **REST API** 并不是说它做了 **REST** 论文中的所有事情，而是它创建了一个服务能够在 **HTTP** 之上进行交互，并且拥有一些像 **REST** 的特性。网络服务（**web services**）和 **REST** 是在 **HTTP** 之上进行构建的方法论，描述了当我们要基于 **HTTP** 搭建新的服务时所需要应用的非常具体的格式和理论。

轮询

客户端从服务器获取到资源的一种方式是轮询（**polling**），即客户端周期性地向服务器发送请求。这样的问题就是，如果服务器发现客户端没有什么好更新的，但是它还是必须要处理这个请求，这就会大量地消耗服务器的网络资源、内存和 **CPU** 资源。所以轮询虽然能让客户端尽可能地得到最新的结果，但是它相应地有非常大的代价。为了解决这个问题，我们可以动态地调整轮询的周期。如果连续的轮询发现没有可以更新的内容，那么我们就指数级地增加轮询的周期，来减轻负担。如果得到了更新的信息，那么我们就降低轮询的周期。

另一种，解决从服务器中得到信息来更新客户端的方法，使用了一种更加新的 **HTTP** 协议，叫做网络套接字协议（**web socket**）

REST

接下来我们来介绍 **REST** 的具体意义，**REST** 从某种意义上来说，代表了一种构建资源 **URL** 的

一种格式。举个例子来说，我们想访问我们视频服务中的视频。一种表示一个具体视频的方法可能是 `/video/:videoid`，如果我们想访问 `video1` 的持续时间，我们可以使用 `/video/1/duration`。如果我们想获取全部的视频列表，我们可以直接访问 `/video`。事实上这就是 REST 的一部分：资源 URL 的部署形式。每加一个 /，就指明了一个更加详细的特性或者当前资源的一个部分。REST 在 http 请求中的设置就是，使用 GET 来获取资源；使用 POST 来追加新资源到尾部；使用 PUT 来新建或者替代已有资源；使用 DELETE 来删除资源。

消息推送

APP 如何实现消息推送呢，主要分成本地推送和在线推送。本地推送可以不需要网络，APP 在执行时向系统注册定时事件，如系统日历等。日历事件触发后，设置对应的 APP 监听回调函数，向主界面发送通知。

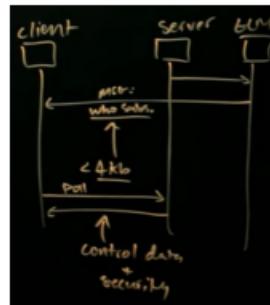
另一种方法是在线推送。安卓 APP 获取在线推送的一种方式是使用 GCM。

GCM，全称为 Google Cloud Messaging，译为 Google 云端通讯。它能够让第三方应用的开发者把通知消息或信息从服务器发送到所有使用这个应用的安卓系统或 Chrome 浏览器的应用或拓展上。

在线推送的三种方式	第一种情况是像 QQ、微信这样的应用，它们长期驻留在系统后台，长期占用一部分内存来推送消息。也就是说，虽然你以为你关闭了 QQ 和微信，但其实这一部分执行推送任务的 QQ 和微信模块仍在后台工作。
	第二种情况是使用第三方推送服务，这是谷歌服务在国内运行情况不佳的产物，第三方推送服务在国内有多家，是在第三方应用中加注 SDK 实现，但在不同的 ROM 中不同的第三方推送能否按时送到则参差不齐。
	第三种情况则是使用了 GCM 服务的应用，主要是大量的来自国外的应用和少部分来自国内的应用，比如 Facebook、Twitter、CNN、BBC，与第三方推送服务不同的是，它是系统层级的，第三方应用的服务器把消息发送给谷歌的服务器再转接到各个用户。IOS、Windows 10 系统的消息推送亦是在应用未运行的情况下由第三方服务器转给苹果或微软的服务器，再推送给用户。

我们具体介绍 GCM 的实现方法。首先，手机会和 GCM 服务器保持 XMPP（基于 XML 的通知协议）长连接。第一次时，手机客户端向 GCM 服务器发送请求，GCM 服务器返回一个唯一的 ID 标识符，然后手机客户端将 ID 标识符发送给需要注册推送的 APP 服务器。APP 服务器得到这个 ID 标识符后，每当需要推送的时候，就通过 POST 请求将需要推送的内容和这个 ID 标识符发送给 GCM 服务器。因为 GCM 服务器和手机客户端保持长连接，GCM 可以将被推送的消息发送给手机。但是 GCM 只允许发送小于 4kb 的信息到客户端，如何发送大量的数据到客户端呢？

我们引入了 a push to sync model，换句话说，我们推送 GCM 信息来通知客户端开始接收数据，接收到 GCM 消息后，客户端向服务器发起轮询请求，然后进行同步传输数据。



这也保护了我们数据的安全，避免一些敏感的数据被谷歌的 GCM 服务直接访问到（我们可以通过 GCM 告诉客户端该和我们自己的服务器建立连接交换数据了）。

课后习题

1. How is metadata about an HTTP request transmitted?

1 分

- URL Encoded Body Parameters
- Path Variables
- JSON Body
- Headers
- Query Parameters

HTTP 请求包含许多元数据信息，如数据详细信息，方法类型，查询参数，主机和端口号等。这些东西都存在 http 请求的 header 中。

2. Which of the following are ways of sending data to a server in an HTTP request?

- Query Parameters
- the Path
- URL Encoded Body Parameters
- JSON Encoded Body

在 http 请求中，客户端向服务器发送数据的方式是：

- | |
|-------------------|
| 1. 直接编码在 URL 字符串中 |
|-------------------|

Query String Parameters

当发起一次GET请求时，参数会以url string的形式进行传递。即?后的字符串则为其请求参数，并以&作为分隔符。
如下http请求报文头：

```
// General  
Request URL: http://foo.com?x=1&y=2  
Request Method: GET  
  
// Query String Parameters  
x=1&y=2
```

2. 使用 URL 编码在请求的 form data 块中

Form Data

当发起一次POST请求时，若未指定content-type，则默认content-type为application/x-www-form-urlencoded，即参数会以Form Data的形式进行传递，不会以键式出现在请求url中。
如下http请求报头：

```
// General  
Request URL: http://foo.com  
Request Method: POST  
  
// Request Headers  
content-type: application/x-www-form-urlencoded; charset=UTF-8  
  
// Form Data  
x=1&y=2
```

3. 使用 JSON 编码在 request payload 中

Request Payload

当发起一次POST请求时，若content-type为application/json，则参数会以Request Payload的形式进行传递（显然，数据格式为JSON），不会以键式出现在请求url中。
如下http请求报头：

```
// General  
Request URL: http://foo.com  
Request Method: POST  
  
// Request Headers  
content-type: application/json; charset=UTF-8  
  
// Request Payload  
{"x":1,"y":2}
```

同样的，不带参数 url 路径本身也定位了资源的位置，提供了请求的信息。所以这道题全选。

3. Which is not an appropriate use of an HTTP header?

- to communicate the type of server responding to a request
- to specify the path for a request
- to communicate the primary payload of an HTTP request
- to communicate the type of client sending the request

The HTTP headers are used to pass additional information between the clients and the server through the request and response header.故选 BC。

4. What are mime types used for?

- to define a schema for the allowed request headers
- to describe the type of data being sent to the client
- to describe the type of data being sent to the server
- to specify the return type of a function
- to define a schema for the allowed response headers

在请求头和响应头的 Content-Type 中，都描述了 body 中的数据的 MIME 类型。故选 BC。

5. Which of the following are components of a communication protocol?

- URLs
- Request Parameters
- Syntax
- Timing
- Semantics

一个通讯协议的组成部分为语法和语义和时间。

6. What are cookies used for?

- to store data needed by the server in the load balancer
- to store data needed by the server in the request headers
- to store data needed by the client on the server
- to store data needed by the server on the client

Cookie 是服务器传给客户端让客户端存在本地的。故选 D

7. What type of protocol is HTTP?

- Request / Response
- Peer to Peer
- Secure
- Onion-routed

选 A

8. Which of the following differentiate query parameters from data sent in the body of a request?

- query parameters can't include addressing parameters
- query parameters cannot be included in a URL
- larger volumes of data are typically sent in the body
- body parameters can be encoded in multiple formats

选 CD。

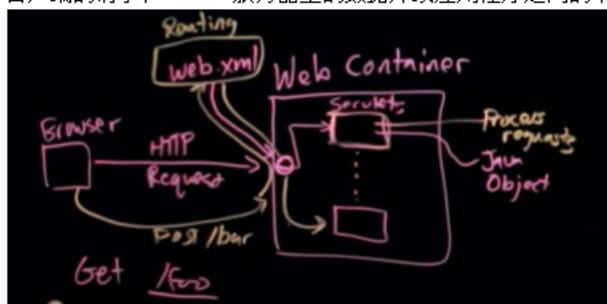
第二单元

Java 如何处理 http 请求

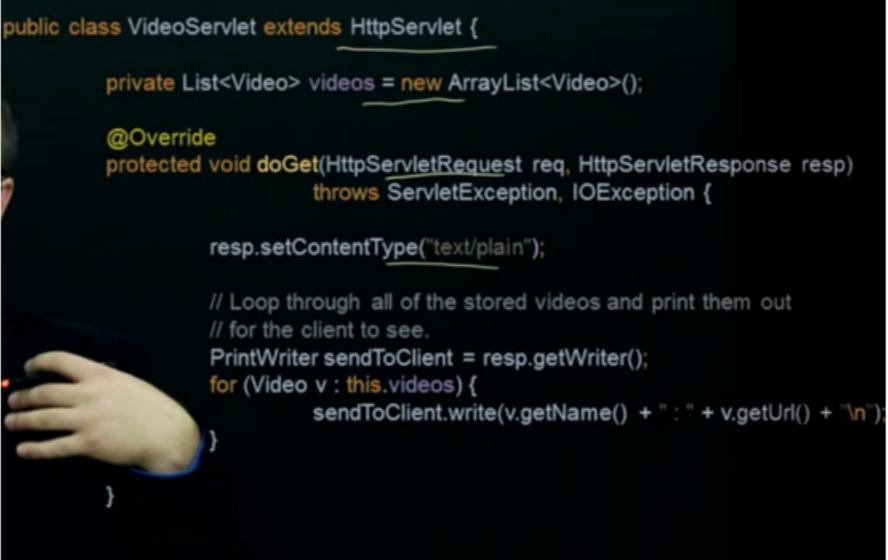
服务器有一个 Web container，其中包含了很多 servlet。

Web 容器是一种服务程序，在服务器一个端口就有一个提供相应服务的程序，而这个程序就是处理从客户端发出的请求，如 JAVA 中的 Tomcat 容器，ASP 的 IIS 或 PWS 都是这样的容器。一个服务器可以有多个容器。

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

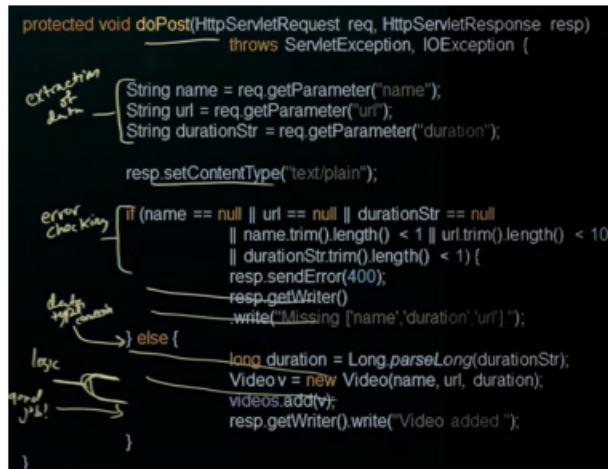


如上图所示，浏览器向服务器发送 http 请求，然后服务器的路由函数从 web.xml 中寻找对应的 Servlet。Servlet 中定义了 doGet、doPost 等函数，来处理对应的请求。



```
public class VideoServlet extends HttpServlet {  
    private List<Video> videos = new ArrayList<Video>();  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
  
        resp.setContentType("text/plain");  
  
        // Loop through all of the stored videos and print them out  
        // for the client to see.  
        PrintWriter sendToClient = resp.getWriter();  
        for (Video v : this.videos) {  
            sendToClient.write(v.getName() + ":" + v.getUrl() + "\n");  
        }  
    }  
}
```

在 `doGet` 函数中，首先设置了响应内容的 `mime type` 是纯文本，然后对于每个视频文件，我们都返回其一个字符串。



```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, IOException {  
  
    String name = req.getParameter("name");  
    String url = req.getParameter("url");  
    String durationStr = req.getParameter("duration");  
  
    resp.setContentType("text/plain");  
  
    if (name == null || url == null || durationStr == null  
        || name.trim().length() < 1 || url.trim().length() < 10  
        || durationStr.trim().length() < 1) {  
        resp.sendError(400);  
        resp.getWriter()  
            .write("Missing [name],[duration],[url] ");  
    } else {  
        long duration = Long.parseLong(durationStr);  
        Video v = new Video(name, url, duration);  
        videos.add(v);  
        resp.getWriter().write("Video added.");  
    }  
}
```

到处理 `post` 请求时，事情就变得比较复杂了。在我们介绍构建在 `servlet` 之上的框架，来给予我们更高层次的抽象之前，我们还是先看一下这个 `doPost` 函数的实现。

首先是提取请求中的参数，然后是对参数进行异常处理，然后是数据类型的转化，最后才是真正的业务逻辑。如果对于每个 `servlet` 都要执行一遍这个代码，会显得非常复杂。我们更希望只关注业务逻辑的代码实现，而不是让提取参数、异常处理、类型转化占据了我们大部分的实现代码。

web.xml



当一个 http 请求到达 web container 以后，要导航到对应的 servlet，就需要 web.xml 承担路由的功能。具体来说，在 web.xml 中分为定义 servlet 和映射路由。

注入攻击

考慮以下一个 echo 服务器，我们客户端上传一些文本內容，我们 echo 服务器的作用就是返回一个 html 响应，对用户上传的文本內容加上 html 的一些组件，如<html><body>客户端上传的文本</body></html>。这时候我們就要小心了，如果客户端上传的文本包含了 JavaScript 代码，那么在返回的网页中可能可以执行这一个恶意的代码来实现某些攻击。我們考慮另一个网站，向 echo 服务器请求，返回一个指向 echo 服务器的链接。如果上传的文本中含有恶意代码，当用户点击这个链接的时候，可能同时执行了 js 恶意代码。如果服务器的返回中有一些敏感的信息，如登录验证的 cookie、或者是银行账户的余额等。可能被这个恶意代码直接转发到第三方服务器上。所以说，为了预防这种注入攻击，我们必须在真正执行业务逻辑前验证上传数据的类型，证明这是一个合法的请求数据而不是一种攻击，然后再去拿这个数据去执行一些敏感的行为。我們不能简单地认为只有我們自己的服务器能发送请求到服务器，事实上任何客户端都可以向我们的服务器发送请求，这时候我們就不能默认请求是友善的，最好我們把所有请求都当做可能有恶意的请求，进行严格的验证校验后，再执行业务逻辑，來保证整个业务的安全性。

Spring 框架中的 Dispatcher Servlet 和 Controller 抽象

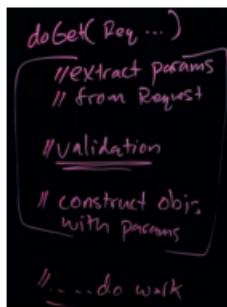
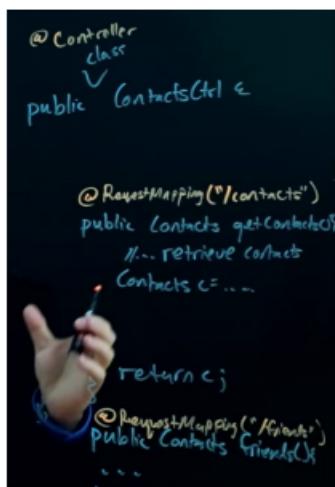


图 1 一些 Servlet 中的 overhead

在之前的课程中，我们提到过：虽然 `servlet` 提供了一个可行的处理 `http` 请求的方法，但是如上图的步骤所示，那些被框起来的步骤都是 `overhead`（间接费用）。为了避免这些开支，我们可以使用 Spring 框架中的 `dispatcher servlet` 来完成这件事情。`Dispatcher Servlet` 是前置控制器，拦截与 `web.xml` 文件中定义匹配的请求。它把拦截下来的请求，依据相应的规则分发到目标 `Controller` 来处理。在 `Controller` 中，比起 `doGet`、`doPost`。可以设置任意的函数，来运行。比如说我们设置了 `update balance(int)` 请求，并且设置 `http` 请求中的 `account-num` 参数映射到这个函数的 `int` 参数中。在执行这个函数之前，会检查 `account-num` 是否是 `int` 类型，然后再进行赋值。这样 `Controller` 就替我们完成了之前需要我们自己完成的操作（从请求中提取参数、验证参数合法性、使用参数创建 Java 对象）。

Spring 中的控制器

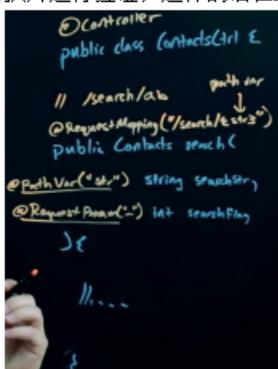


Spring 中的控制器其实就是一个 Java 的类。类前注释 `@Controller`，类内的成员函数如果对应映射一个 `url` 请求的话，使用 `@RequestMapping()` 来实现。

使用 RequestParam 和 PathVar 注释来接受客户端数据

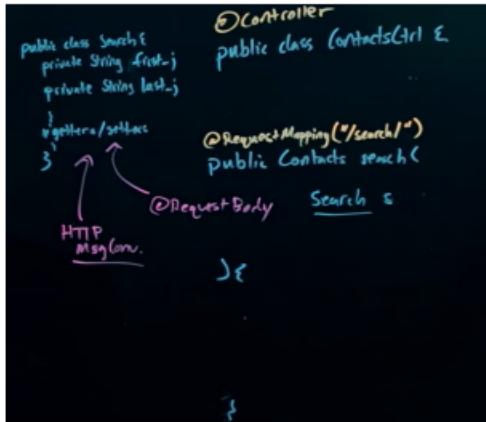


如果我们想路由到一个带参数的函数怎么办？我们在函数形参列表中所定义的参数前加上对应的注释。比如`@RequestParam("search") String searchStr` 就代表了使用 http 请求中所含的参数来初始化函数参数 `searchStr`。`Dispatcher servlet` 会根据 Java 形参的类型自动进行类型转换并进行验证，这样的话在函数体中就只要完成业务逻辑的实现。



以上为从浏览器的路径中提取一些参数。如`@RequestMapping("/searching/{str}")`就是把所有符合这个格式的请求都路由到这个函数中。注意 str 只是一个标识符，并不代表任何的类型。在函数参数定义中，我们使用`@PathVar("str") String searchStr` 来把 str 这个标识符绑定到这个 Java 变量中。

但是，如果每次参数变更了我们就要重新修改函数中的参数以及对应的注释，这会导致代码不可扩展。如果我们能够直接传入某个类作为参数，就可以解决这个扩展性的问题。在 Spring 中，提供了`@RequestBody` 这个注释来修饰函数参数类。当 `dispatcher servlet` 发现这个注释后，会调用 `http message converter` 自动地从 `http` 请求的参数中，转换出相应的 Java 对象，作为参数传进函数中。如下图所示：



SpringMVC——消息转换器 HttpMessageConverter <https://blog.csdn.net/cq1982/article/details/44101293/>

处理 multi-part 数据

到目前为止，我们只是学会上传了一些字符串参数，如何真正上传一个视频文件呢？我们需要使用 multipart，在函数参数中我们设置参数类 MultipartFile。然后在函数实现中，我们创建一个 InputStream 来处理这个二进制文件即可。

```
public class VideoSue {
    public boolean uploadVideo(
        @RequestParam("data")
        MultipartFile videoData
    ) {
        InputStream in = videoData.getInputStream();
        // save it to disk
    }
}
```

为了防止 http 请求中上传了过大的文件导致影响服务器或者对服务器形成攻击，我们需要使用以下的方法配置 multipart 最大上传的字节数以及 http 请求允许的最长字节数。

```
public class Application{  
    add  
    this → {  
        to  
        longi  
    }  
    @Bean  
    public MultipartConfigElement  
        getMultipartConfig(){  
            MultipartConfigFactory f = ...  
            f.setMaxFileSize(2000);  
            f.setMaxRequestSize(...);  
            return f.createMultipartConfig();  
    }  
}
```

Spring MVC 4 使用 Servlet 3 MultiPartConfigElement 实现文件上传（带源码） <https://blog.csdn.net/w605283073/article/details/51340880>

课后习题

1. What is the purpose of request routing?

- to determine which header should be used to authenticate the user
- to determine which function should generate the response for a given request
- to determine which mime type the request should be interpreted as
- to determine which client should receive the response

选 B

2. Which of the following are true of client request data?

- the data could lead to an injection attack
- the data will be in the correct format or the HTTP server would reject it
- the data should be carefully sanitized and validated
- the data will be in the correct format due to Spring annotations

A 正确，客户端请求可以进行攻击注入。B 不正确，服务器在 validation 阶段要手动判断 http 请求中的数据是否合法。C 正确，未经验证的数据可能会进行注入攻击。D 错误，数据不会因为 Spring 框架而从错误的格式变为正确的格式。只是数据本身以正确的格式组织以后，Spring 可以进行格式转换。

3. How is HTTP used in cloud services?

- HTTP can be used as the communication protocol for talking to cloud services
- HTTP provides the data format used inside of services
- HTTP cookies provide the security for cloud applications
- HTTP headers provide routing information for cloud commands

A 是对的；

B 数据格式应当是约定的，似乎不对吧；

C 不对，cookies 就是用来做身份校验的，提升了 cookies 被篡改，非法获取权限的可能性；

D，http 没有提供了路由的信息，是 web.xml 提供的

4. What is the relationship between a Spring controller and the dispatcher servlet?

- controllers inherit from the dispatcher servlet
- the dispatcher servlet routes HTTP requests to one or more controllers to produce a response
- the dispatcher servlet guarantees the security of controllers
- controllers provide the dispatcher servlet with access to a database

选 B

5. Which of the following are true?

- The PathVariable annotation is used to indicate that a method parameter should be bound to a component of the path the HTTP request was sent to
- The RequestBody annotation is used to indicate that a method parameter should be bound to the body of an HTTP request
- The RequestParam annotation is used to indicate that a method parameter should be bound to a specific parameter from the HTTP request

选 A,B,C。

第三单元

使用 `ResponseBody` 注释来创建响应

我们之前讨论了怎么从 `HTTP` 请求中，把对应的数据存到 `JAVA` 对象中。现在我们关注 `HTTP` 的响应部分，也就是怎么把 `JAVA` 对象编码到 `HTTP` 响应中返回给客户端。

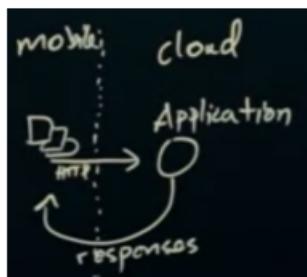
分享

```

@Controller
public class ContactsCtrl {
    @RequestMapping("/search")
    public @ResponseBody Contact
        search(...) {
        //...find contact
        return contact;
    }
}

```

在 Spring 中，这件事情比较容易去做。我们只要在函数返回类型前写上 @ResponseBody，就可以把返回的 JAVA 对象，通过一系列的消息转换器编码进 HTTP 请求中。



自此，我们整个流程就完成了。客户端通过 HTTP 请求发送 commands，互联网上的服务器解析成 JAVA object，执行对应的业务逻辑后，把 JAVA object 通过 message converter 转化为 json 对象作为 HTTP 响应的返回体。

使用 Jackson 框架的注释进行序列化(Marshalling)

Jackson 框架是 Java 中的一个的库，可以让我们快速建立 Java 类与 JSON 之间的关系。我们具体来考察一下，Jackson 框架的序列化是如何实现的。

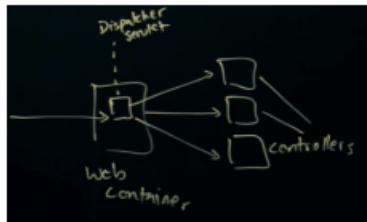
`public class Video {
 private String title;
 private String url;
 private long duration;
 @JsonIgnore
 public String getTitle() {
 return title;
 }
 public void setTitle(String title) {
 this.title = title;
 }
 public String getUrl() {
 return url;
 }
 public void setUrl(String url) {
 this.url = url;
 }
 public long getDuration() {
 return duration;
 }
 public void setDuration(long duration) {
 this.duration = duration;
 }
}`

Annotations and methods corresponding to JSON fields:

- `title`: annotated with `@JsonProperty("title")`, mapped to `getTitle` and `setTitle`.
- `duration`: annotated with `@JsonProperty("duration")`, mapped to `getDuration` and `setDuration`.
- `url`: annotated with `@JsonProperty("url")`, mapped to `getUrl` and `setUrl`.
- `foo`: annotated with `@JsonIgnore`, mapped to `getFoo` and `setFoo`.

如上图所示，我们现在有一个 `Video` 的类。由三个私有成员变量，以及对应的 `get` 和 `set` 函数。一个 `Json` 传来以后，`Jackson` 会自动地先调用类的构造函数，`new` 出一个新的类。然后对 `Json` 里的每个名字，首字母大写，调用类的 `setXXX` 函数，并且根据函数需要参数的类型进行类型转换。如果 `setXXX` 对应的变量本身又是另一个类，则会递归地用 `Json` 的层级结构去初始化。如果 `Json` 多了一个类中没有的变量名，或者类中有一个变量未被初始化，`Jackson` 都会报错。可以在希望忽略的属性上注释 `@JsonIgnore`，这样会忽略这个变量的序列化和反序列化。

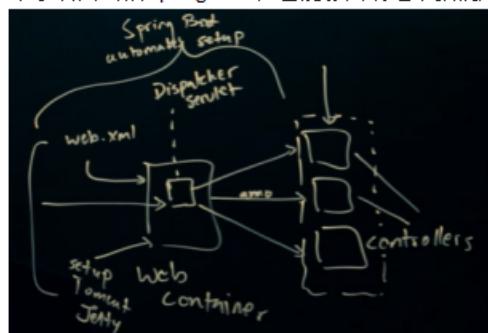
Spring Boot&应用结构



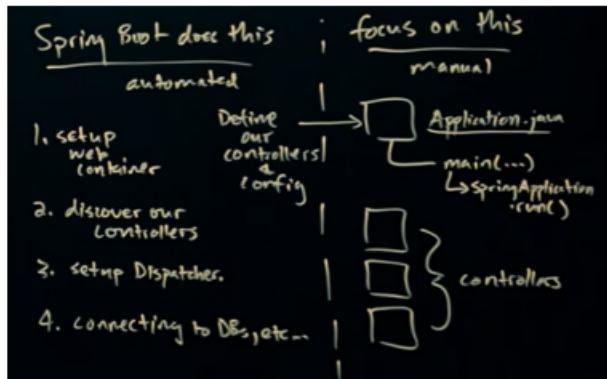
根据目前所学，一个后端的响应结构如上图所示。如果我们要从零开始搭这样的一个架构，就需要 `web.xml` 给 `web` 容器提供路由信息，或者调用一些函数来提供路由信息。

我们需要配置 `web` 容器，比如说 `Tomcat` 或者 `Jetty`，我们需要用 `web.xml` 来安装我们的应用。这些配置都是比较花时间的，我们希望只关注我们业务的部分（`controller` 部分的具体实现）。

所以 `Spring` 有一个子项目叫做 `Spring Boot`，它能够自动地帮我们配置这一系列的东西。



一个 `Spring Boot` 应用有很简单的结构。第一个部分是一系列的业务控制逻辑，也就是路由到的 `controller java` 对象。第二个部分是一个叫做 `Application` 的类，在此之中，我们定义了我们应用的一些配置参数，我们可以在其中可以定义 `controller`，或者告诉 `Spring Boot` 让它自己去搜索我们的 `controller`。



这个 `Application` 类有一个 `public static void main` 函数，启动时，命令行会调用这次函数。这个函数里面会有一些 `SpringApplication.run()` 之类的函数，我们需要把对应的参数传进去。它会替我们做很多事情：1.设置 web 容器 2.找到业务的控制器 3.设置 dispatcher 及其对应的路由 4.连接数据库等。所以我们需要告诉 Spring Boot 去哪里找 controllers 等初始参数。

服务器伸缩

随着用户并发数的上升，我们需要有更多的资源来处理用户的请求。服务器的伸缩（scaling）分为垂直伸缩和水平伸缩。

垂直伸缩是指增大单台服务器的性能，来处理更多的并发，这在 1000 个并发扩展到 10000 个并发这种级别是可行的。但是在超大并发下，我们不难想象，谷歌是找不到一台超级巨大的服务器去处理搜索引擎的所有用户请求。这时候就需要水平伸缩了。

水平伸缩就是指，我们在负载大的时候，新增几台服务器，来增加总体的性能。

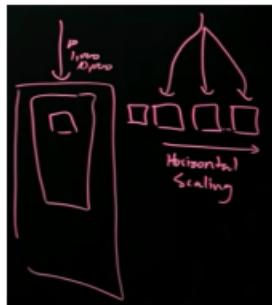
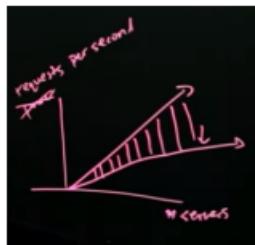


图 2 左：垂直伸缩，右：水平伸缩

我们自然期望随着服务器的增加，单位时间内能够处理的用户请求数随着服务器的数量线性增加。但是这很大程度上取决于分布式服务器的设计，如下图所示：可能理论情况是上面这一条，但是我们实现的是下面这一条线，中间的就是效率的瓶颈。



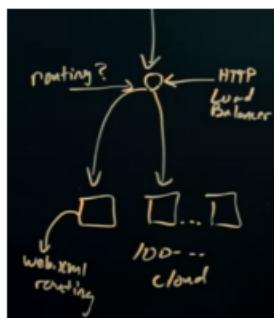
我们希望能够很轻易地添加额外的服务器一个提升效率的方法就是，我们要把我们的应用尽可能设计成无状态的（stateless）。

我们说一个应用是有状态的（stateful），意味着我们需要在服务器端存储一些会话的信息，比如 caches、日志信息等。为了提升服务器的并发，我们除了新开几台机器以外，我们还要处理如何把一些客户端的信息移动到新开的机器上；如何迁移一些重要的状态；新机器如何获得状态访问状态，这就比较麻烦。

但是，如果是一个无状态的应用，我们可以非常容易地新开多台机器来分摊一些请求，不需要考虑客户端信息的迁移，可以非常容易的提升服务器的容量。

所以，无状态性是服务器一个非常重要的特性，我们希望我们的 controller 在不知道发生过什么的情况下就能处理请求。但是这通常是很难达到的。

负载均衡



如果在我们云上有 100 台服务器，怎么样让用户请求分配到不同的服务器上呢？这就需要 HTTP 均衡负载器做这件事情。最简单的策略就是轮询调度策略（round robin scheme），也就是每个服务器轮流接收请求。但是这会要求我们的应用实现是无状态的：举个例子来说，比如客户端 1 和服务器 1 建立了连接，进行了登陆操作。这样服务器 1 上就会有客户端 1 的登录信息，但是如果在下次请求的时候，轮询调度到了服务器 2 上，服务器 2 没有客户端 1 的登录信息，这可能就会要求客户端 1 重新进行登录。这显然是不合理的。一种解决方法是，我们现在的轮询策略是按照“每条请求”这个层次（at individual request level），我们可以上升为“每个客户端”这个层次（at individual client level），同一个客户端的请求我们调整策略使其一定对应到一开始建立连接的那台服务器上（这也叫做粘滞会话 https://blog.csdn.net/tomcat_baby/article/details/52787679）。这就要求我们根据应用是 stateful 还是 stateless 在轮询策略中选取不同的层次。

注：登陆这件事情，如何变成无状态的？

解决登陆这件事情的办法：

把登陆变成无状态的。<https://blog.csdn.net/fhkbbfgggjk/article/details/85047461>

或者采用分布式 `session`，也就是我们把登录信息存在一个中心数据库中，所有服务器都可以通过访问这个数据库来共享登陆信息。其实和上述网址讲的 `token` 验证差不多，因为 `token` 的校验肯定也要从数据库中得到数据对比。

云平台的自动伸缩扩容技术（auto-scaling）

客户请求有平峰之分，按照之前水平扩容的知识。我们在一个 `spike`（短时间请求峰值）来的时候，水平增加很多的虚拟机，等这个峰值过去了，这些多增加的性能就闲置了。同时，增加虚拟机涉及到一系列的成本：购置机器的费用，配置并部署应用程序的成本，服务器闲置时额外的电费开销等。

但是在云服务器中，机器通常是以小时来计费的，同时购买一台新的服务器也不需要这么麻烦，这为自动伸缩扩容技术提供了可能性。我们希望有一个 `intelligence` 来掌管所有的服务器，根据当前客户端的请求量，自动地增加新的服务器或者删除已存在的空闲服务器。这样能够自动地去适应客户请求量的变化。但是在云端部署一台新的服务器，虽然没那么麻烦，也是需要几分钟到几小时不等的。所以这个 `auto-scaling` 技术要对客户请求数进行建模，尽可能预测请求量的变化情况，提前做好增删的决策，这样等到请求峰值过来时，已经有恰好量的服务器做好准备了。

同时，`auto-scaling` 还能监控到服务器集群中死机的服务器，它也可以启动新的服务器来替换掉它。这就要求我们的应用设计是无状态的，这样使用新服务器替代死机的服务器不会带来额外的成本。

IaaS vs. PaaS

当我们把应用部署上云的时候，我们需要考虑使用哪种服务，分为 `IaaS`（Infrastructure as a service）和 `PaaS`（platform as a service）。这是一个很重要的决定，它取决于我们想怎么样部署我们的应用。

`IaaS` 可以理解为，你在云端购买了一个新的机器。你请求云服务器提供商为你开一台虚拟机。然后你会在它之上自己配置应用所需要的不同的组件，比如 `controller`、`utilities`。你可以在节点上搭建自己的数据库。

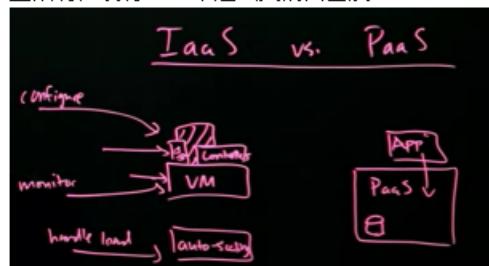
`IaaS` 的优点就是给了你控制虚拟机的类型和参数的灵活性。它允许你控制虚拟机上不同的组件，允许你控制一些涉及安全的事情，比如你可以决定控制器和 `API` 的访问范围。我们可以在虚拟机上自由地建构 `C++` 和 `Java` 环境。

`IaaS` 的缺点就是，必须要自己写脚本来配置、启动、监控这些各种各样的组件。同时，也需要自己去实现 `auto-scaling`（自动伸缩扩展）等功能。所以，`IaaS` 就等于给你了一个裸机，你需要在其之上自己负责所有东西。

而在 `PaaS` 中，只需要提供我们的应用程序，比如一个 `war` 包以及其他处理请求的文件。`PaaS` 得到我们的应用程序以后，会自动为我们配置一个 `web` 容器，配置下层的虚拟机，决

定所有安全的配置，决定何时扩展/收缩（scale it up or down），决定需要多少容量。

PaaS 提供了很多和平台紧耦合的工具链。但是配置的自由度就下降了，我们的应用设计必须符合平台的一些限制，没有 IaaS 中这么大的自由度。



Spring 的依赖注入和自动装配

当我们创建 controller 的时候，他们通过 Spring 自动地实例化。Spring 自动通过注释寻找 controller，并且找出哪些请求要映射到这个 controller 上。但是通常情况下，我们的控制器是依赖于一些我们定义的其他类的。我们可能不想在类中定义这些依赖的类是如何实例化的。我们想能够根据情况的不同，来用不同的方法配置它。

举个例子来说，我们现在有一个控制器叫做 video service，加上 @controller 以后这就是一个简单的 spring 控制器了。比如我们对于我们的视频，有不同的存储选项。我们去定义一个叫做 storage system 的接口 storage 作为成员变量。Storage system 是一个我们在其他地方定义的接口。在 storage system 中可能有不同的实现，比如有 AmazonS3Storage 和 LocalStorage。这就是实现 storage system 接口的不同的类。我们希望我们的控制器能够在不同种类的 storage system 上。

一种方法是，我们简单地实例化一个我们想要使用的特定 storage system。比如说，我们使用 storage = new AmazonS3Storage 或者 storage = new LocalStorage。但是这样的话，每次我们想要更改这个 storage system 的种类，我们都需要重写代码并且重新编译。

我们希望每次使用一些特定配置启动应用的时候，我们所希望的 storage system 类型能够自动地参数化并且注入到控制器中。Spring 提供了实现这个的一种方法，也就是通过依赖注入（dependency injection）。

依赖注入的意思就是，我们可以定义某个类依赖的对象，比如我们的 video service 就依赖于 storage system。依赖注入中注入的部分代表 Spring 可以自动地填入、提供一个对象来执行 storage system 接口，我们不需要构造它。我们实现这个的方法也像魔法一样，我们只需要在这个存储变量上添加一个注释 @AutoWired



这个的意思是，当你想实例化 video service 的时候，你还需要找到一个能够被自动装配或者注入进这个成员变量中的 storage system 对象。Spring 就是检查提供给它的配置，从配置参数中找到 storage system 的实现，然后使用它来创建一个对象的实例，或者重用一个已经存在的对象。然后设置 storage 变量为这个实例的值。它自动地去发现我们的 video service 需要什么实现。并且自动地去找到我们在配置中所定义的实现，最后自动地帮我们设置成员变量。这帮助我们创建了依赖多个对象的 video service。

再举个例子来说，我们可能有另一个依赖对象是在其他地方定义的 UserManager 接口，变量名叫做 user。我们使用@AutoWired 以后，Spring 会自动帮我们去从配置中找到一个实现我们所定义在里面的接口的对象，并且用那个对象来设置我们这里的这个成员变量。它通过依赖注入自动地实例化并且配置好整个对象的层级结构，最终构建出我们的应用。我们可以通过提供不同的配置对象（configuration object）给 Spring 来控制如何去实现它。

为了使用我们对象不同的 AutoWired 特性，我们需要告诉这个依赖注入机制，如何映射到这个独立的接口上。我们就需要使用配置对象（configuration object）。在 Spring 中，我们可以创建一个类来定义不同接口的具体实现（我们的 controller 所依赖的）和我们想要使用的真正具体的类。

为了实现这件事情，我们可以定义一个类叫做 VideoConf，我们用@Configuration 来注释它。这就告诉 Spring 中的依赖注入，要从这个类中找到一些 AutoWired 依赖的映射。为了填东西进 storage system 中，我们可以加一个方法

```
public StorageSystem StorageSystem() {return new LocalStorage();}
```



当你调用这个函数，它就会返回一个 `StorageSystem` 的实现，在这里是 `LocalStorage`，也就是我们希望 `auto-wired` 所依赖的类型。我们需要在方法前添加`@Bean`。在运行过程中，`Spring` 会创建我们 `VideoConf` 对象的实例，并且寻找用`@Bean` 注释的方法，并且会自动地调用这些方法来得到我们希望在接口类中所实现的合适的类。`@beam StorageSystem` 的意思就是我们希望创建一个 `LocalStorage` 的实例，并且在代码里每个存在 `@AutoWired StorageSystem` 的地方，`Spring` 会自动替我们用这个值来设置成员变量。这样我们完全解耦了我们对象的创建和配置。创建一个类，我们只需要注释`@AutoWired`，然后我们在远离具体实现细节的地方，定义这些拥有接口的依赖。为了定义一个特定的应用的用例，我们通过创建`@Configuration` 类，并且在其之中创建`@Bean` 方法来实现不同 `storage system` 的映射。

总体上的实现一开始看起来似乎很复杂，但是这是 Spring 一个很重要的部分，它允许你写一些更加模块化的、可重用的代码。

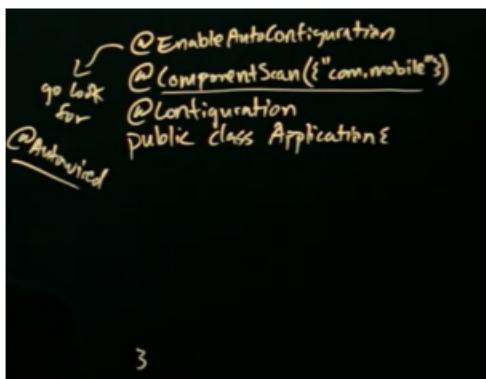
在实际运行过程中，Spring 实例化 video service 的时候，发现了 @AutoWired，它就会去找配置对象中的 @Bean 并且返回合适对象的方法。如果找到了，Spring 就会调用它来得到一个返回值，然后插入成员变量中。Spring 的默认行为是只会调用 @Bean 对应的函数一次。也就是它只会创建 LocalStorage 的一个实例，然后每个 @AutoWired 对应的 StorageSystem 都会引用一个相同的对象。当然也可以改变这个行为，比如每次看到 @AutoWired 都去调用一次这个方法，得到一个新的实例对象插入。你不需要使用复杂的机制来传递所需要的对象了，你只需要使用 Spring 的这个依赖注入，自动地使用 @Configuration 中的 @Bean 的对应方法插入 @AutoWired 所注释的成员变量。

当你使用配置通过 Spring 的依赖注入来自动构建应用的时候，Spring 提供了注释来简化任务，比如`@Configuration` 以及`@Bean`。但是，在只有一个接口实现的时候，仍有很多观众。比所有实现都在一个相同的包里。如果你告诉 Spring 去自动去找变量接口的实现，并且自动把这些接口和实现联系在一起，它会使用它自己找到的实现。

你可以通过`@ComponentScan`告诉 Spring, Java 包的名字, 或者 Java 包的前缀, 比如 `com.mobile`。这时候 Spring 就回去扫描这个包, 找到所有接口。比如说它找到了一个 `@AutoWired` 的实现, 它会使用这个包里的找到的实现去插入进成员变量里。

这节课我们假设是基于成员扫描包的一一映射。因为如果它扫描了包并且发现了一个接

口的多种实现，Spring 不能自动地选择一个进行自动装配。



另一个重要的注释`@EnableAutoConfiguration`，它告诉 Spring 去自动找那些`@Autowired`的注释，并且自动填入。如果我们不加上这个注释，Spring 不会自动去寻找。通常情况下，我们会看见`@EnableAutoConfiguration` 和 `@ComponentScan`一起使用。使用这两个注释可以简化我们的配置类，因为我们不需要再去为每一个组件定义`@Bean`以及对应的方法。值得注意的是，`@ComponentScan("com.mobile")`会递归地扫描目录下的所有 Java 包，如果内容很多的话，会消耗比较多的时间。但是这是在每次启动时所消耗的，只要我们服务开的足够久，那就没什么问题。但是有一些云服务的提供商，比如 Google APP Engine 会在没有请求时自动关闭服务，这就会对性能有比较大的影响，因为我们每次启动都要用 ComponentScan 去扫描一次包，把所有的`AutoWired`变量都装配好。

1. Which of the following are true of dependency injection? 1分
- it is a form of cyber-attack stemming from improper sanitization of client data
 - none of the above
 - it requires users of a dependency injection library to write code to pass dependencies to consumers of the dependencies
 - it is best used when there is exactly one implementation of each interface
2. What is horizontal scaling? 1分
- increasing the width parameter in an HTTP request
 - increasing the number of hosts supporting a cloud application
 - increasing the CPU power (e.g., a better CPU) of the hosts supporting a cloud application
 - adding load balancers to a cloud application
3. Which of the following would make horizontal scaling more difficult? 1分
- having cookies that store state data needed by the server
 - having a load balancer distributed requests in a "sticky" fashion
 - having state that persists across requests from a client
 - having state used by all functions on a host that service a given request
4. What is the purpose of the `ResponseBody` annotation? 1分
- to indicate which controller method should be invoked to produce the HTTP response body
 - to indicate which view associated with a controller should produce the HTTP response body
 - to indicate which member variable should be used to create the HTTP response body
 - to indicate that the return value from a method should be transformed into the HTTP response body
5. Which of the following are examples of IaaS? 1分
- providing the ability to provision network addresses to a cloud service
 - providing the ability to provision messaging queues to a cloud service
 - providing the ability to provision virtual machines to a cloud service
 - providing the ability to provision database tables to a cloud service

课后习题

1. Which of the following are true of dependency injection?
- it is a form of cyber-attack stemming from improper sanitization of client data
 - none of the above
 - it requires users of a dependency injection library to write code to pass dependencies to consumers of the dependencies
 - it is best used when there is exactly one implementation of each interface

选 B

2. What is horizontal scaling?

1分

- increasing the width parameter in an HTTP request
- increasing the number of hosts supporting a cloud application
- increasing the CPU power (e.g., a better CPU) of the hosts supporting a cloud application
- adding load balancers to a cloud application

选 B

3. Which of the following would make horizontal scaling more difficult?

1分

- having state that persists across requests from a client
- having a load balancer distributed requests in a "sticky" fashion
- having cookies that store state data needed by the server
- having state used by all functions on a host that service a given request

选 A

4. What is the purpose of the `ResponseBody` annotation?

1分

- to indicate which view associated with a controller should produce the HTTP response body
- to indicate which member variable should be used to create the HTTP response body
- to indicate that the return value from a method should be transformed into the HTTP response body
- to indicate which controller method should be invoked to produce the HTTP response body

选择 C

5. Which of the following are examples of IaaS?

1分

- providing the ability to provision database tables to a cloud service
- providing the ability to provision messaging queues to a cloud service
- providing the ability to provision virtual machines to a cloud service
- providing the ability to provision network addresses to a cloud service

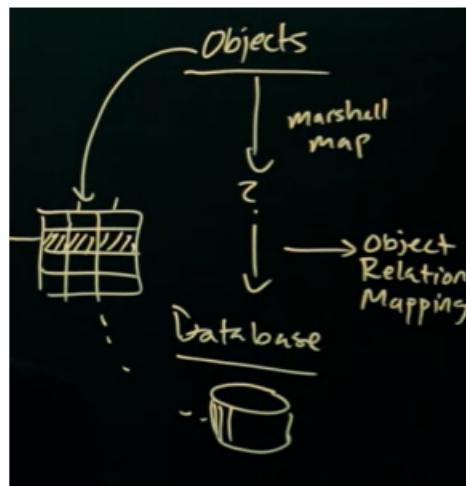
AB 错误

C 错误

IaaS 得到的是逻辑，

第四单元

数据库映射



我们想在服务器上以类似的方法存储一些数据。在服务器端，我们有对象，我们希望从对象中提取出一些信息，并且转化为数据库中存储的某种个数。数据库中存储的数据可能有多种格式和类型。不过都是需要把我们的对象映射到数据库的格式中。正如 Jackson 对 Json 和 Java 对象做的转化一样。在数据库中，数据可能是以行和列的方式存储的。我们就需要知道如何把 Java 对象映射到数据库中的行和列。通常我们需要自己写代码，输入每个对象，描述对象是如何映射到数据库表中的。比如说，我们可能对每个对象映射到数据库的一行里。然后每一列描述了对象一个具体的属性或一个成员变量。我们可以把成员变量放在列中，把对象实例映射到不同的行中。但是如果这样做的话，每次保存一个对象，我们都要手动指出如何把对象转化为数据库中表的格式；每次从数据库中取出数据，我们还要指出如何从数据库的表中取出数据转化回 Java 对象。这整个过程就叫做对象关系映射，至少对于传统的关系型数据库来说是这样的。

这个属于在非关系型数据库中，比如 noSQL 数据库中，也适用。基本的事实就是：当我们需要存储的时候，需要在服务器端把对象转化为某种数据库可以理解的格式；当我们需要使用数据的时候，我们需要找到数据转化为对象，然后再进行业务逻辑。因为 Java 只能在对象上进行操作，我们不希望在数据库的原始表上直接进行操作。

Java 持久化 API (JPA)

现在我们的 video 存储在 VideoService 类里，这会导致如果应用重启，保存的用户上传的数据都会丢失。所以，我们要把 video 数据可持久化地存储到数据库里。正如上一节课“数据库映射”所提到的，在 Java 中，提供了直接帮助我们创建数据库映射的接口，我们可以免去一些手写复杂的数据库映射操作。

```
@Table(name="CUSTOMERS")
@Entity
public class Customer {
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Id
    private Integer id;
    private String name;
    private String email;
    private int age;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
}

@Entity
public class Video {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String name;
}
```

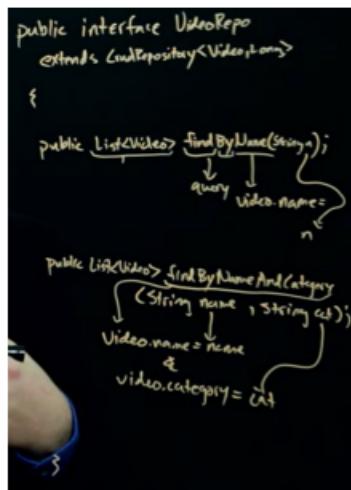
`@Entity` 用来注释所需要存储的类，其中必须要有一个`@Id` 注释来标识能够唯一确定一个实例的主键。为了省去我们自己去设置 `Id` 生成策略的麻烦，JPA 提供了一个注释：`@GeneratedValue`

`@GeneratedValue`：
`@GeneratedValue` 用于标注主键的生成策略，通过`strategy` 属性指定。默认情况下，JPA 自动选择一个最适合底层数据库的主键生成策略：SqlServer 对应 identity，MySQL 对应 auto increment。
在`javax.persistence.GenerationType`中定义了以下几种可供选择的策略：
-IDENTITY：采用数据库ID自增长的方式来自增主键字段，Oracle 不支持这种方式；
-AUTO：JPA自动选择合适的策略，是默认选项；
-SEQUENCE：通过序列产生主键，通过`@SequenceGenerator` 注解指定序列名，MySQL不支持这种方式
-TABLE：通过表产生主键，框架借由表模拟序列产生主键，使用该策略可以使应用更易于数据库移植。

Spring 的仓库

现在我们创建了一系列用`@Entity`注释的类，并且给定了其实例唯一的`ID`。我们接下来需要通过 JPA 真正和数据库进行交互，执行对对象的存储、查询、取值操作。

Spring 提供了仓库(repository)来做这件事，我们在 Spring 中可以创建一个仓库的接口。要创建我们的 video 仓库，我们首先需要一个 `public interface`，叫做 `VideoRepo`，它扩展了 `CurdRepository`。我们需要用我们正在处理的类型 `Video, Long` 来参数化 `CurdRepository`。其中 `Video` 代表了我们要存储的数据类，`Long` 代表了其中 `Id` 的类型。我们定义这个接口为了更新、取回我们的 `Video` 对象。Spring 框架会自动地为我们填入这个接口的实现，不需要我们自己去处理。这个接口允许我们通过一些优美的机制来和数据库交互。注意到，`CrudRepository` 是标准的仓库接口，`CRUD` 代表了 `create, read, update, delete`。默认地，我们自动从 `CrudRepository` 继承了一系列的方法。我们可以通过填入 `Id` 的方法来查找对象。通过扩展 `CrudRepository`，我们得到了一些基础的操作来在数据库中存储对象、查询符合参数条件的对象等，这些 Spring 都帮我们做了。



一个我们所需要的最重要的函数就是查询符合某些标准的 `Video` 类。比如说，我们想查询符合某个名字的 `Video` 类。我们只需要为我们的接口定义新的方法。我们可以添加一个方法 `public List<Video> findByName (String n)`。这个函数名的定义必须符合格式，`find` 是指查询，`Name` 和参数中的 `n` 是指要查询的是所有符合 `Video.name = n` 的 `Video`，返回一个列表。Spring 自动地创建了这个接口函数的实现，其能够查询所有的 `Video` 对象，并且返回符合条件的对象列表。

同样地，如果我们要查询符合两个条件的，即可以定义 `public List<Video>findByNameAndCatagory (String name, String cat)`，Spring 就会去查找 `Video.name = name` 并且 `Video.catagory = cat` 的对象。

通过简单地给我们的接口定义新的方法（以 `find` 开头），并且提供一系列的参数。我们就能够在数据库中根据参数进行查询，Spring 为我们自动地去实现查询和查找方法。当然还有很多进阶的查询技巧，我们可以从 `Spring data JPA` 文档，或者 `Repository` 文档中去找。

SQL注入攻击

```
public class VideoService {
    select * from
    video where
    owner = 'Larsen'
    and name = 'sql'
}

? → foo' or 'a'='a
select * from video
where [owner = 'foo'
and name = 'foo']or 'a'='a'
↑
new logic

public List<Video> myVideos(String owner)
User u = getUser(...);

String query =
    "select * from video
     where owner = " +
     u.getName() +
    "' and name = '" +
    name + "');"

return execute(query);
}
```

如上图所示，如果有一个函数直接从客户端得到一个 `String`，然后把它拼接进 SQL 查询语句中。这是一件非常危险的事情，如左下角的例子所示。如果用户的字符串输入 `foo' or 'a'='a'`。这会导致在 SQL 语句中添加进了新的逻辑，导致 SQL 查询会返回所有存储的 `Video` 类，无论是不是这个用户所拥有的。包括用户还可以执行修改任意数据、删除任意数据。所以当我们写查询语句的时候，一定要非常小心，不能让客户端有机会控制我们的逻辑。如果我们使用 `Spring data` 或者 `Spring data rest`，使用 `findBy` 和 `findAll` 等这些函数的话，它会自动构建查询语句，这会保证应用对这些注入攻击免疫。如果你从更低层级去构建 SQL 查询语句，我们就必须意识到注入攻击的可能性，必须小心地构建查询语句。

Spring data 代码

```
8  /**
9   * An interface for a repository that can store Video
10  * objects and allow them to be searched by title.
11  *
12  * @author jules
13  *
14  */
15 @Repository
16 public interface VideoRepository extends CrudRepository<Video, Long>{
17
18     // Find all videos with a matching title (e.g., Video.name)
19     public Collection<Video> findByName(String title);
20
21     public Collection<Video> findByDuration(long duration);
```

如上图所示，我们创建一个 VideoRepository 接口。

```
10
11 //Tell Spring to automatically inject any dependencies that are marked in
12 //our classes with @Autowired
13 @EnableAutoConfiguration
14 // Tell Spring to automatically create a JPA implementation of our
15 // VideoRepository
16 @EnableJpaRepositories(basePackageClasses = VideoRepository.class)
17 // Tell Spring that this object represents a Configuration for the
18 // application
19 @Configuration
20 // Tell Spring to turn on WebMVC (e.g., it should enable the DispatcherServlet
21 // so that requests can be routed to our Controllers)
22 @EnableWebMvc
23 // Tell Spring to go and scan our controller package (and all sub packages) to
24 // find any Controllers or other components that are part of our application.
25 // Any class in this package that is annotated with @Controller is going to be
26 // automatically discovered and connected to the DispatcherServlet.
27 @ComponentScan
28 public class Application {
29
30     // Tell Spring to launch our app!
31     public static void main(String[] args) {
32         SpringApplication.run(Application.class, args);
33     }
34
35 }
```

需要在主函数前添加@EnableJpaRepositories

3. 完整的@EnableJpaRepositories注解

复制代码

```
1 @EnableJpaRepositories(  
2     basePackages = {},  
3     basePackageClasses = {},  
4     includeFilters = {},  
5     excludeFilters = {},  
6     repositoryImplementationPostfix = "Impl",  
7     namedQueriesLocation = "", //META-INF/jpa-named-queries.properties  
8     queryLookupStrategy=QueryLookupStrategy.Key.CREATE_IF_NOT_FOUND, //QueryLookupStrategy.Key.X  
9     repositoryFactoryBeanClass=JpaRepositoryFactoryBean.class, //class  
10    entityManagerFactoryRef="entityManagerFactory",  
11    transactionManagerRef="transactionManager",  
12    considerNestedRepositories=false,  
13    enableDefaultTransactions=true  
14 )
```

2) basePackageClasses

指定 Repository 类

```
1 @EnableJpaRepositories(basePackageClasses = BookRepository.class)  
  
1 @EnableJpaRepositories(  
2     basePackageClasses = {ShopRepository.class, OrganizationRepository.class})
```

备注：测试的时候发现，配置包类的一个Repositories类，该包内其他Repositories也会被加载

```
public class VideoSvc implements VideoSvcApi {  
  
    // The VideoRepository that we are going to store our videos  
    // in. We don't explicitly construct a VideoRepository, but  
    // instead mark this object as a dependency that needs to be  
    // injected by Spring. Our Application class has a method  
    // annotated with @Bean that determines what object will end  
    // up being injected into this member variable.  
    //  
    // Also notice that we don't even need a setter for Spring to  
    // do the injection.  
    //  
    @Autowired  
    private VideoRepository videos;  
  
    // Receives POST requests to /video and converts the HTTP  
    // request body, which should contain json, into a Video  
    // object before adding it to the list. The @RequestBody  
    // annotation on the Video parameter is what tells Spring  
    // to interpret the HTTP request body as JSON and convert  
    // it into a Video object to pass into the method. The  
    // @ResponseBody annotation tells Spring to convert the  
    // return value from the method back into JSON and put  
    // it into the body of the HTTP response to the client.  
    //  
    // The VIDEO_SVC_PATH is set to "/video" in the VideoSvcApi  
    // interface. We use this constant to ensure that the  
    // client and service paths for the VideoSvc are always  
    // in sync.  
    //
```

然后，我们在 controller 中添加 VideoRepository，并用@Autowired 注释。这样控制器就会自己去找 VideoRepository 的实现。

```

@RequestMapping(value=VideoSvcApi.VIDEO_SVC_PATH, method=RequestMethod.POST)
public @ResponseBody boolean addVideo(@RequestBody Video v){
    videos.save(v);
    return true;
}

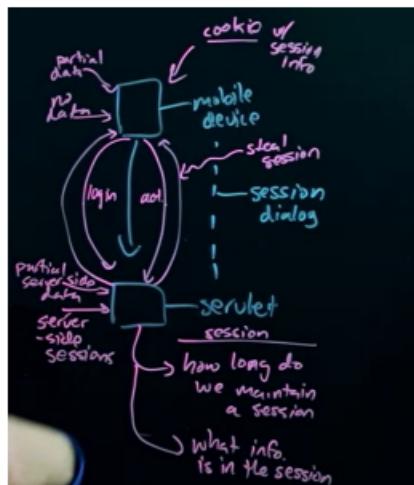
// Receives GET requests to /video and returns the current
// list of videos in memory. Spring automatically converts
// the list of videos to JSON because of the @ResponseBody
// annotation.
@RequestMapping(value=VideoSvcApi.VIDEO_SVC_PATH, method=RequestMethod.GET)
public @ResponseBody Collection<Video> getVideoList(){
    return Lists.newArrayList(videos.findAll());
}

```

从上图中可以看见，videos 是 VideoRepository 的实例，继承了 CrudRepository 接口的函数。

会话（Sessions）

比如说，我们现在实现了一个银行存款查询系统。客户端第一个请求发过来，执行登陆；第二个请求发过来，希望查询存款。这时候我们就需要会话机制，维护这个会话，以免客户端再登陆一次。比如手机端突然断网了几秒，或者失去连接了几秒，我们仍希望之后能够继续进行新的请求。Session 也分为几种方式，服务器可以存储会话的全部信息，也可以存储部分信息，把剩下的部分信息以 cookie 的形式存到客户端，注意这部分 cookie 通常是客户端难以破解的、不太敏感的信息，比如会话的 ID。同时，服务器端也要设置每个会话的超时时间，因为维持一个会话本身就要消耗服务器的一定资源。



Spring Data REST

```
@Controller  
public class VideoSave  
  
    @Autowired  
    private VideoRepo repo  
  
    @RequestMapping("video/save")  
    public void save(Video v){  
        → repo.save(v);  
    }  
  
    @DeleteMapping("video/{id}")  
    public void delete(@PathVariable Long id)  
  
    @GetMapping("video/{id}")  
    public Video findbyId(@PathVariable Long id)
```

我们发现在很多函数中，`controller` 只是提供了请求映射，然后把任务交给了 `VideoRepository` 去做，比如存储数据、获取数据、删除数据等。我们还要花很大功夫为每一个逻辑去创建这一系列的方法。我们希望 `Spring` 提供了一个方法，能够自动地把 `VideoRepository` 作为一个服务暴露给 `HTTP` 请求。客户端可以直接和 `Spring` 中的 `VideoRepository` 交互，我们不需要写下非常重复性的代码，我们希望消解掉这些重复性的工作，希望能够交给 `Spring` 自己去做。

`Spring` 的确提供了简化整个流程的方法，叫做 `Spring Data Rest`。这里面的基本逻辑就是我们可以让一个仓库暴露给一系列的请求映射，比如 `/video` 或 `/video/{property}`。我们可以通过不同的 `HTTP` 请求方法来对这些资源执行动作。因为 `Spring` 通过 `Spring Data Rest` 提供了基于 `rest` 的机制来和仓库交互。我们把仓库视为一个 `rest` 的资源，`GET` 请求就是从资源中取值。`Put` 就是替代资源。

```
    // @RequestMapping(value=VideoSvcApi.VIDEO_SVC_PATH, method=RequestMethod.POST)
    public @ResponseBody boolean addVideo(@RequestBody Video v){
        videos.save(v);
        return true;
    }

    // Receives GET requests to /video and returns the current
    // list of videos in memory. Spring automatically converts
    // the list of videos to JSON because of the @ResponseBody
    // annotation.
    @RequestMapping(value=VideoSvcApi.VIDEO_SVC_PATH, method=RequestMethod.GET)
    public @ResponseBody Collection<Video> getVideoList(){
        return Lists.newArrayList(videos.findAll());
    }

    // Receives GET requests to /video/find and returns all Videos
    // that have a title (e.g., Video.name) matching the "title" request
    // parameter value that is passed by the client
    @RequestMapping(value=VideoSvcApi.VIDEO_TITLE_SEARCH_PATH, method=RequestMethod.GET)
    public @ResponseBody Collection<Video> findByTitle{
        // Tell Spring to use the "title" parameter in the HTTP request's query
        // string as the value for the title method parameter
        @RequestParam(TITLE_PARAMETER) String title
    }{
        return videos.findByName(title);
    }
```

如上图所示，我们看之前实现的 controller，发现里面还是有一些冗余的东西，比如调用一个 @RequestMapping 后我们要手动执行一个简单的 Repo 类的函数并返回。Spring 可以帮我们简化这个操作。

```
// This @RepositoryRestResource annotation tells Spring Data Rest to
// expose the VideoRepository through a controller and map it to the
// "/video" path. This automatically enables you to do the following:
//
// 1. List all videos by sending a GET request to /video
// 2. Add a video by sending a POST request to /video with the JSON for a video
// 3. Get a specific video by sending a GET request to /video/{videoId}
//   (e.g., /video/1 would return the JSON for the video with id=1)
// 4. Send search requests to our findByXYZ methods to /video/search/FindByXYZ
//   (e.g., /video/search/findByName?title=Foo)
//
@RepositoryRestResource(path = VideoSvcApi.VIDEO_SVC_PATH)
public interface VideoRepository extends CrudRepository<Video, Long>{

    // Find all videos with a matching title (e.g., Video.name)
    public Collection<Video> findByName(
        // The @Param annotation tells Spring Data Rest which HTTP request
        // parameter it should use to fill in the "title" variable used to
        // search for Videos
        @Param(VideoSvcApi.TITLE_PARAMETER) String title);

    // Find all videos that are shorter than a specified duration
    public Collection<Video> findByDurationLessThan(
        // The @Param annotation tells Spring Data Rest which HTTP request
        // parameter it should use to fill in the "duration" variable used to
        // search for Videos
        @Param(VideoSvcApi.DURATION_PARAMETER) long maxduration);

    /*
     * See: http://docs.spring.io/spring-data/jpa/docs/1.3.0.RELEASE/reference/html/jpa.repositories.html
     * for more examples of writing query methods
     */
}
```

如上图所示，我们可以使用 @RepositoryRestResource 注释来创建一个 Restful 服务，而无需自定义控制器并手写一系列的函数。构建 RESTful 服务（使用 Spring Data JPA）
https://blog.51cto.com/u_13501268/2404421

这个服务提供了接口自带的一系列函数（见上图绿色注释），免去了一系列自己实现

的麻烦。我们利用越多的 Spring 的特性，我们的代码实现就会越简单。因为 Spring 去帮我们实现了复杂的函数，并且保证了其正确性。