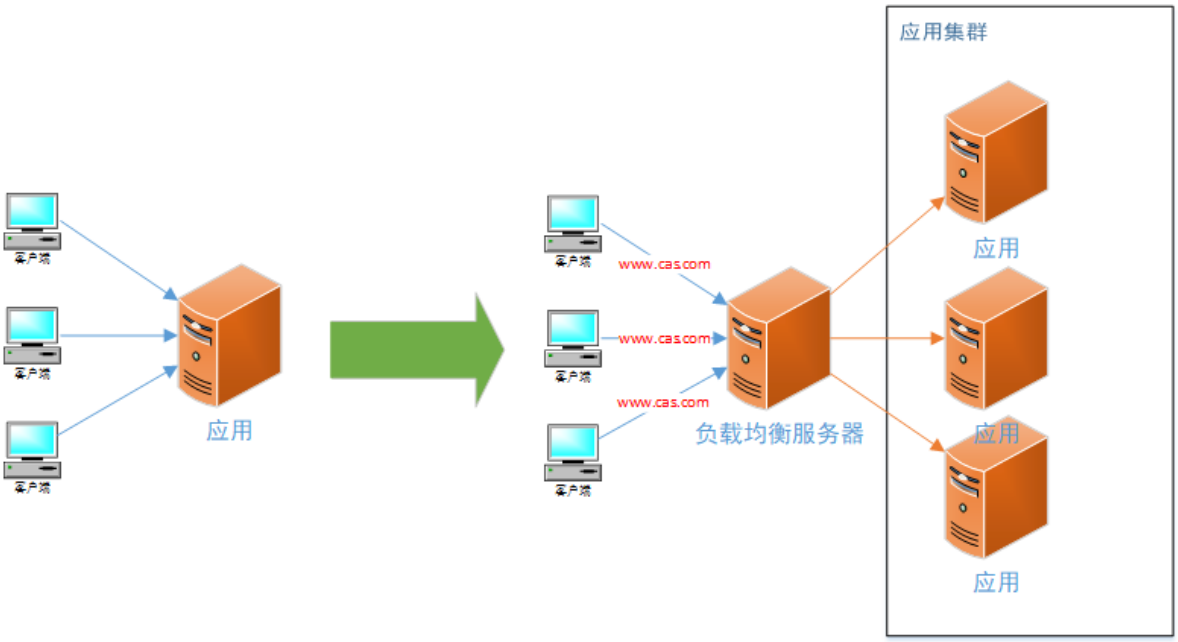
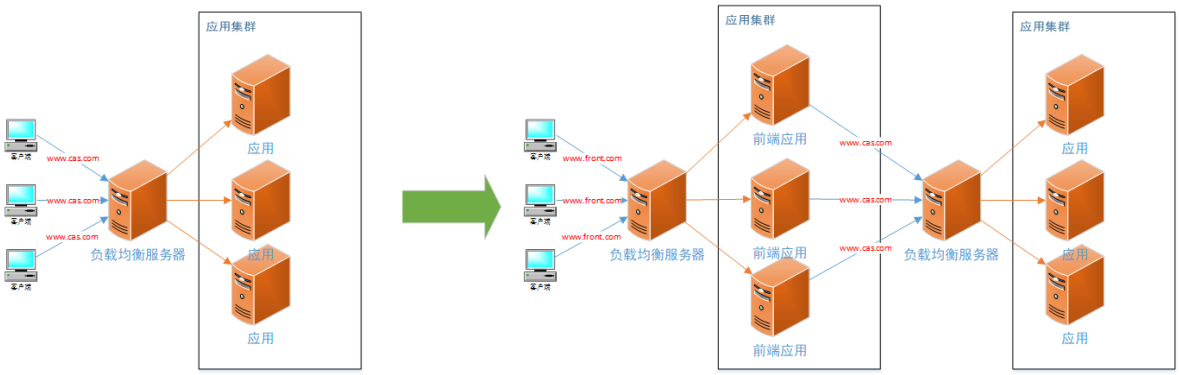
跨域问题的解决方案

1.背景介绍

传统的应用采用单体部署的方式，应用集成了所有的功能，作为一个整体进行部署，如下所示。随着业务量的增长，单应用已无法及时有效处理高并发的用户请求，此时，单应用所能处理的并发请求量成为应用的性能瓶颈。为了缓解单应用的性能瓶颈，系统开始横向扩展，采用集群的方式部署应用，利用多台部署相同应用的机器一起分担并发的用户请求，从而缓解系统的性能瓶颈，适应用户请求量的增长。



后续随着微服务架构思想的传播以及静态资源和动态资源加载速度不一致,静态资源可加速传播等特点，系统开始采用了前后端应用的架构分离，形成了现有的一种架构方式。



跨域资源共享(CORS，Cross-Origin Resource Sharing) 是一种机制，它使用额外的 HTTP 头来告诉浏览器 让运行在一个 origin (domain) 上的Web应用被准许访问来自不同源服务器上的指定的资源。当一个资源从与该资源本身所在的服务器**不同的域、协议或端口**请求一个资源时，资源会发起一个**跨域 HTTP 请求。**

> CORS需要浏览器和服务器的支持，这个通信过程由浏览器自动完成。目前，主流浏览器都兼容

简单来说，浏览器一发现请求是一个**跨域请求**时，需要**判断请求的类型**

简单请求和非简单请求请查阅此处HTTP访问控制(CORS)

如果是**简单请求**，会在请求头中增加一个 **Origin** 字段，表示这次请求是来自哪一个**源。**而服务器接受到请求后，会返回一个响应，响应头中会包含一个叫 **Access-Control-Allow-Origin** 的字段，它的值**要么包含由 Origin 首部字段所指明的域名，要么是一个 "*"**，表示接受任意域名的请求。如果响应头中没有这个字段，就说明当前源不在服务器的许可范围内，浏览器就会报错:

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/access-control/simpleXSInvocation.html
Origin: http://foo.example
```

如果是**非简单请求**，在通信之前，发送一个**预检请求（preflight）**，目的在于询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些 HTTP 动词和头信息字段，只有得到肯定答复，浏览器才会发出正式的请求，否则就报错。

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST                                    // 告知服务器，实际
请求将使用 POST 方法
```

```
Access-Control-Request-Headers: X-PINGOTHER, Content-Type        // 知服务器，实际请
求将携带两个自定义请求首部字段：
// INGOTHER 与 Content-Type

// 响应
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS         //  服务器允许客户端使用
POST, GET 和 OPTIONS 方法发起请求
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type // 服务器允许请求中携带字段
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

出于安全原因，浏览器限制从脚本内发起的跨源HTTP请求。 例如，XMLHttpRequest和Fetch API遵循同源策略。 这意味着使用这些API的Web应用程序只能从加载应用程序的同一个域请求HTTP资源，除非响应报文包含了正确CORS响应头。

示例：

URL	说明	是否允许通信
http://www.test.com/a http://www.test.com/b http://www.test.com/c	同一域名，不同的访问路径	允许
http://www.test.com:8080/a http://www.test.com:9001/a	同一域名，不同端口，跨域	不允许
http://www.test.com/a https://www.test.com/a	同一域名，不同协议，跨域	不允许
http://a.test.com/a http://b.test.com/a	主域名相同，子域名不同	不允许
http://www.test.com http://www.hello.com	域名不同	不允许

跨域资源共享标准新增了一组 HTTP 首部字段，允许服务器声明哪些源站通过浏览器有权限访问哪些资源。另外，规范要求，**对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 `GET` 以外的 HTTP 请求，或者搭配某些 MIME 类型的 `POST` 请求），浏览器必须首先使用 `OPTIONS` 方法发起一个预检请求（preflight request）**，从而获知服务端是否允许该跨域请求。服务器确认允许之后，才发起实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 Cookies 和 HTTP 认证相关数据）。

2.解决方案

2.1 通过jsonp跨域

- 实现思路：

  1. 本站的脚本创建一个元素，src 地址指向跨域请求数据的服务器
  2. 提供一个回调函数来接受数据，函数名可以通过地址参数传递进行约定
  3. 服务器收到请求后，返回一个包装了 JSON 数据的响应字符串，类似这样：callback({...})

```
function jsonp({ url, params, callback }) {
  return new Promise((resolve, reject) => {
    // 创建一个临时的 script 标签用于发起请求
    const script = document.createElement('script');
    // 将回调函数临时绑定到 window 对象，回调函数执行完成后，移除 script 标签
    window[callback] = data => {
      resolve(data);
      document.body.removeChild(script);
    };
    // 构造 GET 请求参数，key=value&callback=callback
    const formatParams = { ...params, callback };
    const requestParams = Object.keys(formatParams)
      .reduce((acc, cur) => {
        return acc.concat([`${cur}=${formatParams[cur]}`]);
      }, [])
      .join('&');
    // 构造 GET 请求的 url 地址
    const src = `${url}?${requestParams}`;
    script.setAttribute('src', src);
    document.body.appendChild(script);
  });
}

// 调用时
jsonp({
  url: 'https://xxx.xxx',
  params: {...},
  callback: 'func',
})
// 我们用 Promise 封装了请求，使异步回调更加优雅，但是别看楼上的洋洋洒洒写了一大段，其实本质上就是：
<script src='https://xxx.xxx.xx?key=value&callback=xxx'><script>
```

- 特点
  - 优点： 简单而且兼容性很好
  - 缺点：**需要服务器支持而且只支持 GET 请求**

2.2 使用spring中的@CrossOrigin注解

- 对控制类中的所有方法有效

```
@RequestMapping("/user")
@CrossOrigin(allowCredentials = "true", allowedHeaders = "*")
public class LoginController {

    @PostMapping("login")
    public JsonResult login() {
        return JsonResult.ok().data("token", "admin");
    }
```

```
    @GetMapping("info")
    public JsonResult info(){
        return JsonResult.ok()
        .data("roles","[admin]")
        .data("name","amdin")
        .data("avatar","https://www.xxx.com/f778738c-e4f8-4870-b634-
56703b4acafe.gif");
    }
}
```

- 仅对单个方法有效

```
@RequestMapping("/user")
public class LoginController {
    @CrossOrigin(allowCredentials = "true", allowedHeaders = "*")
    @PostMapping("login")
    public JsonResult login() {
        return JsonResult.ok().data("token", "admin");
    }

    @GetMapping("info")
    public JsonResult info(){
        return JsonResult.ok()
        .data("roles","[admin]")
        .data("name","amdin")
        .data("avatar","https://www.xxx.com/f778738c-e4f8-4870-b634-
56703b4acafe.gif");
    }
}
```

注意:

使用@CrossOrigin注解后仍然不能跨域访问的解决方法

情形一：在方法上使用@CrossOrigin注解，但@RequestMapping注解中没有指定Get、Post方式，通过method = RequestMethod.POST/GET指定后，问题解决。

情形二：跨域访问响应状态码是405-Method Not Allowed，请求行中指定的请求方法不能被用于请求相应的资源。原因很明显，就是请求不正确，检查代码，使用正确的方式请求。

情形三：查看**springboot版本，如果是2.0以后版本，allowCredentials属性的默认值为false**，返回的响应头AccessControlAllowCredentials属性值也为false，如果**客户端携带cookie的请求**这时**是不能跨域访问的**，所以需要手动在注解中设置allowCredentials为true 即@CrossOrigin(allowCredentials = “true”)

2.3 使用拦截器方式

- Java注解方式，通过实现**WebMvcConfigurer** 接口

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
```

```
        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

- XML配置方式

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="https://domain1.com, https://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="true"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="https://domain1.com" />

</mvc:cors>
```

- 使用Spring Security方式

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.applyPermitDefaultValues();
        config.setAllowCredentials(true);
//        config.addAllowedOrigin("https://domain1.com");       // 配置发起请求的位
置
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");

        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);

        CorsFilter filter = new CorsFilter(source);
      return filter;
    }
}
```

2.4 使用过滤器方式

- 使用filter过滤器

```
// /* 表示全部拦截
@WebFilter(filterName = "loginFilter", urlPatterns = "/*")
public class CORSFilter implements Filter {
    //这里面 填写不需要 被拦截的地址
    private static final Set<String> ALLOWED_PATHS =
Collections.unmodifiableSet(
            new HashSet<String>(Arrays.asList("/login"))
    );

    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;

        //解决跨域的问题
        response.setHeader("Access-Control-Allow-Origin", "*");  //可以根据不同环境
进行适配
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.setHeader("Access-Control-Allow-Headers", "Content-
Type,Content-Length, Authorization, Accept,X-Requested-With,X-App-Id, X-Token");
        response.setHeader("Access-Control-Allow-Methods",
"PUT,POST,GET,DELETE,OPTIONS");
        response.setHeader("Access-Control-Max-Age", "3600");

        if (request.getMethod().equals("OPTIONS")) {
            response.getWriter().println("ok");
            return;
        }
        String path =
request.getRequestURI().substring(request.getContextPath().length()).replaceAll(
"[/]+$", "");
       boolean allowePath = ALLOWED_PATHS.contains(path);
        if (!allowePath) {  //需要拦截的方法
           Object aa = request.getSession().getAttribute("user");
            if (aa != null) {
                chain.doFilter(request, response);
            } else {
                response.getWriter().write("noLogin");
            }
        } else {  //不需要被拦截的方法
            //直接放行
            chain.doFilter(request, response);
        }

    }

     @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void destroy() {

    }
}
```

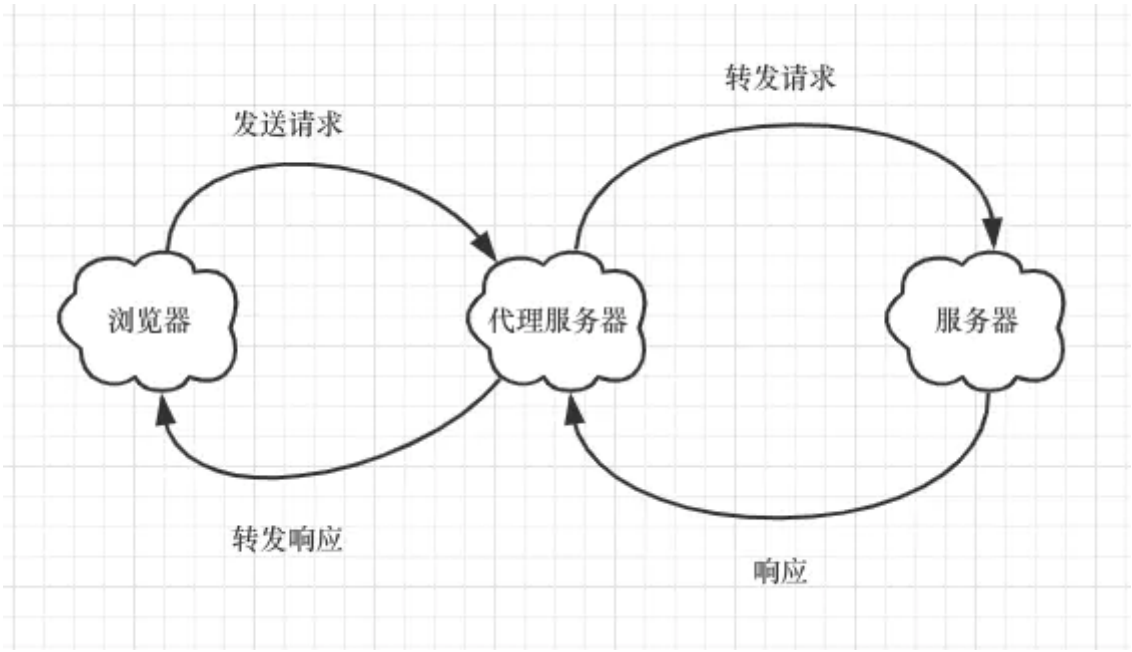
2.4 Websocket

Websocket 是 HTML5 的一个持久化的协议，它**实现了浏览器与服务器的全双工通信**，同时也是跨域的一种解决方案。什么是全双工通信？ 简单来说，**就是在建立连接之后，server 与 client 都能主动向对方发送或接收数据。**

原生的WebSocket API使用起来不方便，需要进行封装，或者使用第三方的库

2.5 Nginx反向代理

原理： 服务器发送跨域请求时不受浏览器的同源策略限制。利用Nginx代理服务器进行请求的转发，进行实现跨域请求。




```
# nginx.config
# ...
server {
  listen       80;
  server_name  www.domain1.com;              // 请求该域名的请求将被转发到
www.domain2.com
  location / {
    proxy_pass   http://www.domain2.com:8080;  #反向代理
    proxy_cookie_domain www.domain2.com www.domain1.com; #修改cookie里域名
    index  index.html index.htm;

    # 当用 webpack-dev-server 等中间件代理接口访问 nignx 时，此时无浏览器参与，故没有同源限制，下面的跨域配置可不启用
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Credentials true;
    # ...
  }
}
```

2.6 Node中间件代理

与2.4节的原理相同，利用Node中间件作为代理。

```
const https = require('https')
// 接受客户端请求
const sever = https.createServer((req, res) => {
  ...
  const { method, headers } = req
  // 设置 CORS 允许跨域
  res.writeHead(200, {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Methods': '*',
    'Access-Control-Allow-Headers': 'Content-Type',
    ...
  })
  // 请求服务器
  const proxy = https.request({ host: 'xxx', method, headers, ...}, response =>
{
    let body = ''
    response.on('data', chunk => { body = body + chunk })
    response.on('end', () => {
      // 响应结果转发给客户端
      res.end(body)
    })
  })
  // 结束请求
  proxy.end()
})
```

附录

1. 简单请求

简单请求指不会触发**CORS预检**的请求。若请求满足以下所有条件，则该请求为简单请求：

- 使用以下方法之一

  GET、HEAD、POST

- 除了被用户代理自动设置的首部字段（如： Connection，User-Agent）和在Fetch规范中定义为禁用首部明后才能的其他首部，允许人为设置的字段为 Fetch 规范定义的 对 CORS 安全的首部字段集合。该集合为：
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` （需要注意额外的限制）
  - `DPR`
  - `Downlink`
  - `Save-Data`
  - `Viewport-Width`
  - `Width`

- `Content-Type` 的值仅限于下列：
  - `text/plain`
  - `multipart/form-data`

    - `application/x-www-form-urlencoded`
    - `application/json`
- 请求中的任意 `XMLHttpRequestUpload` 对象均没有注册任何事件监听器；`XMLHttpRequestUpload` 对象可以使用 `XMLHttpRequest.upload` 属性访问。
- 请求中没有使用 `ReadableStream` 对象。

2. 非简单请求

所有不符合简单请求条件的请求

3. 预检请求

“需预检的请求”要求必须首先使用 `OPTIONS` 方法发起一个预检请求到服务器，以获知服务器是否允许该实际请求。"预检请求“的使用，可以避免跨域请求对服务器的用户数据产生未预期的影响。

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST                             // 告知服务器，实际
请求将使用 POST 方法
Access-Control-Request-Headers: X-PINGOTHER, Content-Type       // 知服务器，实际请
求将携带两个自定义请求首部字段：
// INGOTHER 与 Content-Type

// 响应
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS        //  服务器允许客户端使用
POST, GET 和 OPTIONS 方法发起请求
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type // 服务器允许请求中携带字段
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

4. 携带身份凭证的请求

一般来说，对于跨域 `XMLHttpRequest` 或 Fetch 请求，浏览器**不会**发送身份凭证信息。如果要发送凭证信息，需要设置 `XMLHttpRequest` 的某个特殊标志位。

下面的例子中，http://foo.example 的某脚本向 http://bar.other 发起一个GET 请求，并设置 Cookies：

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain(){
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;          // 允许向服务器发送cookie，如果该属性设置为true，则后端不能使用‘*’
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}

// 如果服务器端的响应中未携带 Access-Control-Allow-Credentials: true ，浏览器将不会把响应内容返回给请求的发送者。
```

客户端与服务器端交互示例如下：

```
GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2
Access-Control-Allow-Origin: http://foo.example
```

```
Access-Control-Allow-Credentials: true                    // 缺少该属性，则响
应内容不会返回给请求的发起者。
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain


[text/plain payload]
```

5. 身份凭证的请求与通配符

对于附带身份凭证的请求，服务器不得设置 `Access-Control-Allow-Origin` 的值为“`*`”。

对于不需要携带身份凭证的请求，服务器可以指定该字段的值为通配符，表示允许来自所有域的请求。

这是因为请求的首部中携带了 `Cookie` 信息，如果 `Access-Control-Allow-Origin` 的值为“`*`”，请求将会失败。而将 `Access-Control-Allow-Origin` 的值设置为 `http://foo.example`，则请求将成功执行。

另外，响应首部中也携带了 Set-Cookie 字段，尝试对 Cookie 进行修改。如果操作失败，将会抛出异常。

6. 过滤器和拦截器的区别

- 过滤器： 依赖于servlet容器。在实现上基于函数回调，可以对几乎所有请求进行过滤。使用过滤器的目的是用来做一些过滤操作，获取我们想要获取的数据。
- 拦截器：依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架。在实现上基于Java的反射机制，属于面向切面编程（AOP）的一种运用。由于拦截器是基于web框架的调用，因此可以使用Spring的依赖注入（DI）进行一些业务操作，同时一个拦截器实例在一个controller生命周期之内可以多次调用。但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求则没办法进行拦截处理

相关参考链接

1. HTTP访问控制(CORS)
2. 可能是最好的跨域解决方案