

1.NIO

1.1 NIO通道客户端【应用】

- 客户端实现步骤
 1. 打开通道
 2. 指定IP和端口号
 3. 写出数据
 4. 释放资源
- 示例代码

```
public class NIOClient {  
    public static void main(String[] args) throws IOException {  
        //1.打开通道  
        SocketChannel socketChannel = SocketChannel.open();  
  
        //2.指定IP和端口号  
        socketChannel.connect(new InetSocketAddress("127.0.0.1",10000));  
  
        //3.写出数据  
        ByteBuffer byteBuffer = ByteBuffer.wrap("一点寒毛先制".getBytes());  
        socketChannel.write(byteBuffer);  
  
        //4.释放资源  
        socketChannel.close();  
    }  
}
```

1.2 NIO通道服务端【应用】

- NIO通道
 - 服务端通道
只负责建立建立，不负责传递数据
 - 客户端通道
建立建立并将数据传递给服务端
 - 缓冲区
客户端发送的数据都在缓冲区中
 - 服务端通道内部创建出来的客户端通道
相当于客户端通道的延伸用来传递数据
- 服务端实现步骤
 1. 打开一个服务端通道
 2. 绑定对应的端口号
 3. 通道默认是阻塞的，需要设置为非阻塞

4. 此时没有门卫大爷，所以需要经常看一下有没有连接发过来没？
5. 如果有客户端来连接了，则在服务端通道内部，再创建一个客户端通道，相当于是客户端通道的延伸
6. 获取客户端传递过来的数据，并把数据放在byteBuffer1这个缓冲区中
7. 给客户端回写数据
8. 释放资源

- 示例代码

```
public class NIOServer {
    public static void main(String[] args) throws IOException {
        // 1.打开一个服务端通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        // 2.绑定对应的端口号
        serverSocketChannel.bind(new InetSocketAddress(10000));
        // 3.通道默认是阻塞的，需要设置为非阻塞
        //如果传递true 表示通道设置为阻塞通道...默认值
        //如果传递false 表示通道设置为非阻塞通道
        serverSocketChannel.configureBlocking(false);
        // 4.此时没有门卫大爷，所以需要经常看一下有没有连接发过来没？
        while (true) {
            // 5.如果有客户端来连接了，则在服务端通道内部，再创建一个客户端通道，相当于是客户端通道的延伸
            //此时已经设置了通道为非阻塞
            //所以在调用方法的时候，如果有客户端来连接，那么会创建一个SocketChannel对象。
            //如果在调用方法的时候，没有客户端来连接，那么他会返回一个null
            SocketChannel socketChannel = serverSocketChannel.accept();
            //System.out.println(socketChannel);
            if(socketChannel != null){
                // 6.客户端将缓冲区通过通道传递给服务端，就到了这个延伸通道socketChannel里面
                // 7.服务端创建一个空的缓冲区装数据并输出
                ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                //获取传递过来的数据，并把他们放到byteBuffer缓冲区中。
                //返回值：
                //正数：表示本次读到的有效字节个数。
                //0    ：表示本次没有读到有效字节。
                //-1   ：表示读到了末尾
                int len = socketChannel.read(byteBuffer);
                System.out.println(new String(byteBuffer.array(),0,len));
                //8.释放资源
                socketChannel.close();
            }
        }
    }
}
```

1.3 NIO通道练习【应用】

- 客户端

- 实现步骤

1. 打开通道
2. 指定IP和端口号

3. 写出数据
 4. 读取服务器写回的数据
 5. 释放资源
- 示例代码

```
public class Clinet {
    public static void main(String[] args) throws IOException {
        // 1.打开通道
        SocketChannel socketChannel = SocketChannel.open();
        // 2.指定IP和端口号
        socketChannel.connect(new InetSocketAddress("127.0.0.1",10000));
        // 3.写出数据
        ByteBuffer byteBuffer1 = ByteBuffer.wrap("吃俺老孙一棒棒".getBytes());
        socketChannel.write(byteBuffer1);
        // 手动写入结束标记
        socketChannel.shutdownOutput();

        System.out.println("数据已经写给服务器");
        // 4.读取服务器写回的数据
        ByteBuffer byteBuffer2 = ByteBuffer.allocate(1024);
        int len;
        while((len = socketChannel.read(byteBuffer2)) != -1){
            byteBuffer2.flip();
            System.out.println(new String(byteBuffer2.array(),0,len));
            byteBuffer2.clear();
        }
        // 5.释放资源
        socketChannel.close();
    }
}
```

- 服务端

- 实现步骤
 1. 打开一个服务端通道
 2. 绑定对应的端口号
 3. 通道默认是阻塞的，需要设置为非阻塞
 4. 此时没有门卫大爷，所以需要经常看一下有没有连接发过来没？
 5. 如果有客户端来连接了，则在服务端通道内部，再创建一个客户端通道，相当于是客户端通道的延伸
 6. 获取客户端传递过来的数据，并把数据放在byteBuffer1这个缓冲区中
 7. 给客户端回写数据
 8. 释放资源
- 示例代码

```
public class Sever {
    public static void main(String[] args) throws IOException {
        // 1, 打开一个服务端通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        // 2, 绑定对应的端口号

        serverSocketChannel.bind(new InetSocketAddress(10000));
```

```

// 3, 通道默认是阻塞的, 需要设置为非阻塞
serverSocketChannel.configureBlocking(false);
// 4, 此时没有门卫大爷, 所以需要经常看一下有没有连接发过来没?
while(true){
    // 5, 如果有客户端来连接了, 则在服务端通道内部, 再创建一个客户端通道, 相当于是客
    户端通道的延伸
    SocketChannel socketChannel = serverSocketChannel.accept();
    if(socketChannel != null){
        System.out.println("此时有客户端来连接了");
        // 6, 获取客户端传递过来的数据, 并把数据放在byteBuffer1这个缓冲区中
        ByteBuffer byteBuffer1 = ByteBuffer.allocate(1024);
        //socketChannel.read(byteBuffer1);
        int len;
        //针对于缓冲区来讲
        //如果 从添加数据 ----> 获取数据 flip
        //如果 从获取数据 ----> 添加数据 clear
        while((len = socketChannel.read(byteBuffer1)) != -1){
            byteBuffer1.flip();
            System.out.println(new String(byteBuffer1.array(),0,len));
            byteBuffer1.clear();
        }

        System.out.println("接收数据完毕, 准备开始往客户端回写数据");
        // 7, 给客户端回写数据
        ByteBuffer byteBuffer2 = ByteBuffer.wrap("哎哟, 真疼啊!!!".getBytes());
        socketChannel.write(byteBuffer2);
        // 8, 释放资源
        socketChannel.close();
    }
}
}
}
}

```

1.4 NIO通道练习优化【应用】

- 存在问题

服务端内部获取的客户端通道在读取时, 如果读取不到结束标记就会一直阻塞

- 解决方案

将服务端内部获取的客户端通道设置为非阻塞的

- 示例代码

```

// 客户端
public class CClient {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();

        socketChannel.connect(new InetSocketAddress("127.0.0.1", 10000));

        ByteBuffer byteBuffer1 = ByteBuffer.wrap("吃俺老孙一棒棒".getBytes());
        socketChannel.write(byteBuffer1);
    }
}

```

```

        System.out.println("数据已经写给服务器");

        ByteBuffer byteBuffer2 = ByteBuffer.allocate(1024);
        int len;
        while((len = socketChannel.read(byteBuffer2)) != -1){
            System.out.println("客户端接收回写数据");
            byteBuffer2.flip();
            System.out.println(new String(byteBuffer2.array(),0,len));
            byteBuffer2.clear();
        }
        socketChannel.close();
    }
}

// 服务端
public class Sever {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

        serverSocketChannel.bind(new InetSocketAddress(10000));

        serverSocketChannel.configureBlocking(false);

        while(true){
            SocketChannel socketChannel = serverSocketChannel.accept();
            if(socketChannel != null){
                System.out.println("此时有客户端来连接了");
                // 将服务端内部获取的客户端通道设置为非阻塞的
                socketChannel.configureBlocking(false);
                //获取客户端传递过来的数据,并把数据放在byteBuffer1这个缓冲区中
                ByteBuffer byteBuffer1 = ByteBuffer.allocate(1024);
                //socketChannel.read(byteBuffer1);
                int len;
                //针对于缓冲区来讲
                //如果 从添加数据 ----> 获取数据 flip
                //如果 从获取数据 ----> 添加数据 clear
                while((len = socketChannel.read(byteBuffer1)) > 0){
                    System.out.println("服务端接收发送数据");
                    byteBuffer1.flip();
                    System.out.println(new String(byteBuffer1.array(),0,len));
                    byteBuffer1.clear();
                }

                System.out.println("接收数据完毕,准备开始往客户端回写数据");

                ByteBuffer byteBuffer2 = ByteBuffer.wrap("哎哟,真疼啊!!!".getBytes());
                socketChannel.write(byteBuffer2);

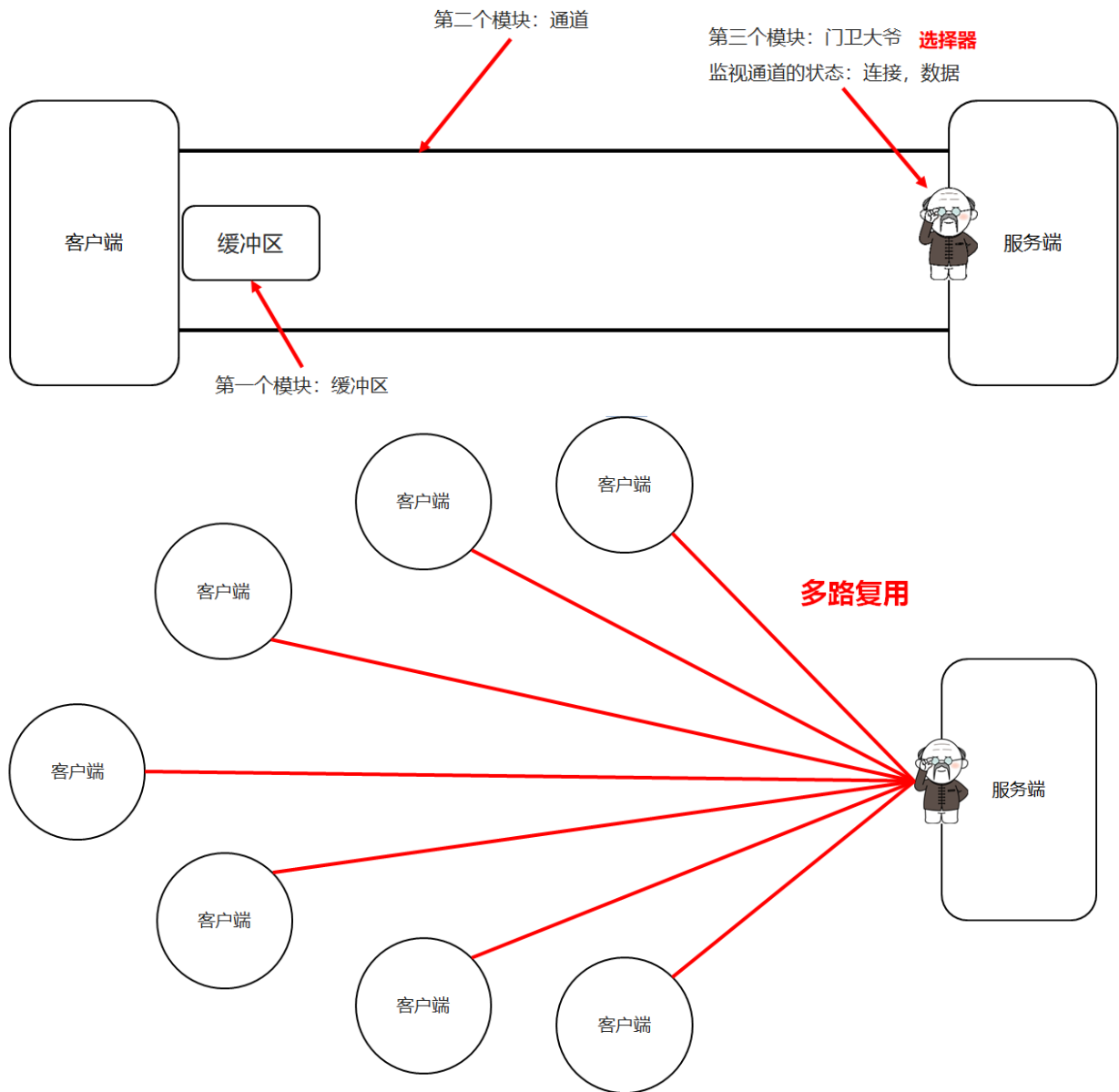
                socketChannel.close();
            }
        }
    }
}

```

1.5NIO选择器【理解】

- 概述

选择器可以监视通道的状态,多路复用



- 选择器对象

- Selector

选择器对象

- SelectionKey

绑定的key

- SelectableChannel

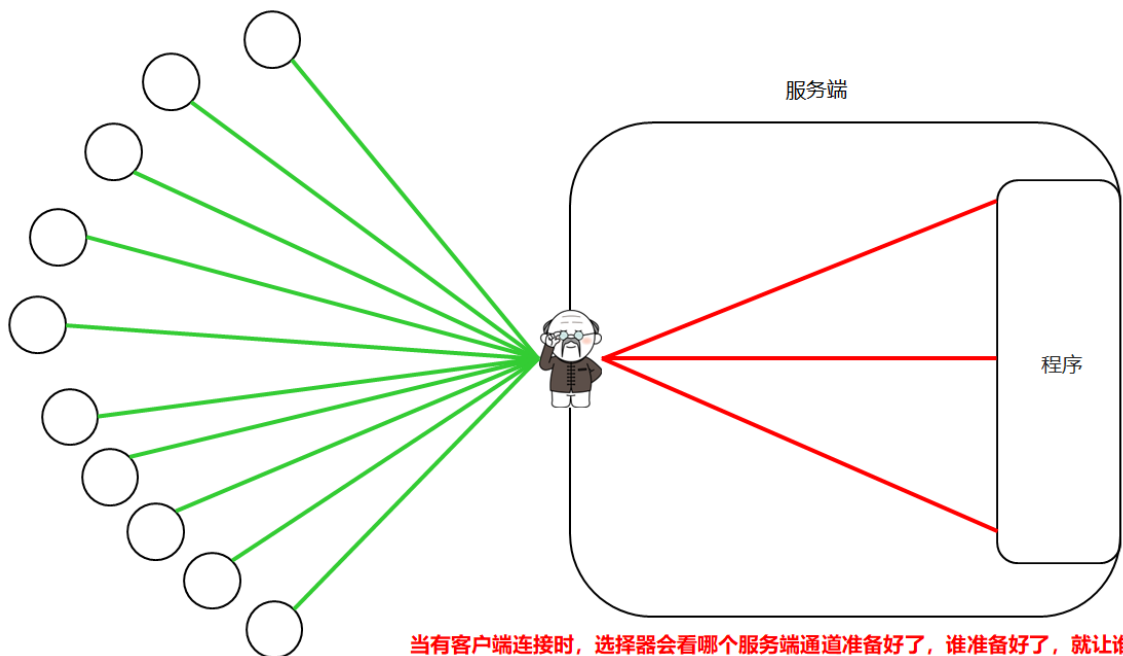
能使用选择器的通道

- SocketChannel
- ServerSocketChannel

1.6NIO选择器改写服务端【应用】

- 实现步骤

1. 打开一个服务端通道(open)
2. 绑定对应的端口号
3. 通道默认是阻塞的，需要设置为非阻塞
4. 打开一个选择器（门卫大爷）
5. 将选择器绑定服务端通道，并监视服务端是否准备好
6. 如果有客户端来连接了，大爷会遍历所有的服务端通道，谁准备好了，就让谁来连接 连接后，在服务端通道内部，再创建一个客户端延伸通道
7. 如果客户端把数据传递过来了，大爷会遍历所有的延伸通道，谁准备好了，谁去接收数据



- 代码实现

```
// 客户端
public class Clinet {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();

        socketChannel.connect(new InetSocketAddress("127.0.0.1",10000));

        ByteBuffer byteBuffer1 = ByteBuffer.wrap("吃俺老孙一棒棒".getBytes());
        socketChannel.write(byteBuffer1);

        System.out.println("数据已经写给服务器");

        ByteBuffer byteBuffer2 = ByteBuffer.allocate(1024);
        int len;
        while((len = socketChannel.read(byteBuffer2)) != -1){
            System.out.println("客户端接收回写数据");
            byteBuffer2.flip();
            System.out.println(new String(byteBuffer2.array(),0,len));

            byteBuffer2.clear();
        }
    }
}
```

```

    }
    socketChannel.close();
}
}
// 服务端
public class Server {
    public static void main(String[] args) throws IOException {
        //1.打开服务端通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        //2.让这个通道绑定一个端口
        serverSocketChannel.bind(new InetSocketAddress(10000));
        //3.设置通道为非阻塞
        serverSocketChannel.configureBlocking(false);
        //4.打开一个选择器
        //Selector --- 选择器
        //    SelectionKey --- 绑定通道后返回那个令牌
        //    SelectableChannel --- 可以使用选择器的通道
        Selector selector = Selector.open();
        //5.绑定选择器和服务端通道
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        while(true){
            System.out.println("11");
            //选择器会监视客户端通道的状态.
            //6.返回值就表示此时有多少个客户端来连接.
            int count = selector.select();
            System.out.println("222");
            if(count != 0){
                System.out.println("有客户端来连接了");
                //7.会遍历所有的服务端通道.看谁准备好了,谁准备好了,就让谁去连接.
                //获取所有服务端通道的令牌,并将它们都放到一个集合中,将集合返回.
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while(iterator.hasNext()){
                    //selectionKey 依次表示每一个服务端通道的令牌
                    SelectionKey selectionKey = iterator.next();
                    if(selectionKey.isAcceptable()){
                        //可以通过令牌来获取到了一个已经就绪的服务端通道
                        ServerSocketChannel ssc = (ServerSocketChannel)
selectionKey.channel();
                        //客户端的延伸通道
                        SocketChannel socketChannel = ssc.accept();
                        //将客户端延伸通道设置为非阻塞的
                        socketChannel.configureBlocking(false);
                        socketChannel.register(selector, SelectionKey.OP_READ);
                        //当客户端来连接的时候,所有的步骤已经全部执行完毕.
                    }else if(selectionKey.isReadable()){
                        //当前通道已经做好了读取的准备(延伸通道)
                        SocketChannel socketChannel = (SocketChannel)
selectionKey.channel();
                        ByteBuffer byteBuffer1 = ByteBuffer.allocate(1024);
                        //socketChannel.read(byteBuffer1);

                        int len;

```



```

        while((len = socketChannel.read(byteBuffer1)) > 0){
            byteBuffer1.flip();
            System.out.println(new String(byteBuffer1.array(),0,len));
            byteBuffer1.clear();
        }
        //给客户端的回写数据
        socketChannel.write(ByteBuffer.wrap("哎哟喂好疼啊!!!".getBytes()));
        socketChannel.close();
    }
    iterator.remove();
}
}
}
}
}
}
}
}
}
}

```

2.HTTP协议

2.1概述【理解】

超文本传输协议(关于超文本的概念JavaWeb在进行学习), 是建立在TCP/IP协议基础上,是网络应用层的协议。

由请求和响应构成,是一个标准的客户端和服务端模型

2.2URL【理解】

- 概述

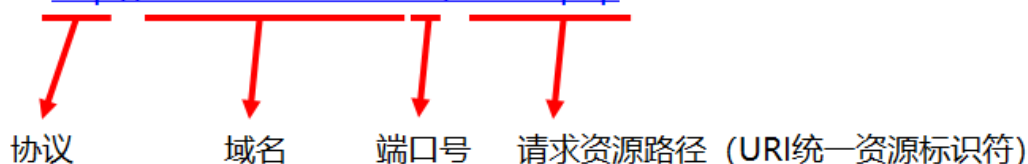
统一资源定位符,常见的如<http://bbs.itheima.com/forum.php>

完整的格式为 <http://bbs.itheima.com:80/forum.php>

- 详解

网址: <http://bbs.itheima.com/forum.php> → URL (统一资源定位符)

完整的格式为: <http://bbs.itheima.com:80/forum.php>

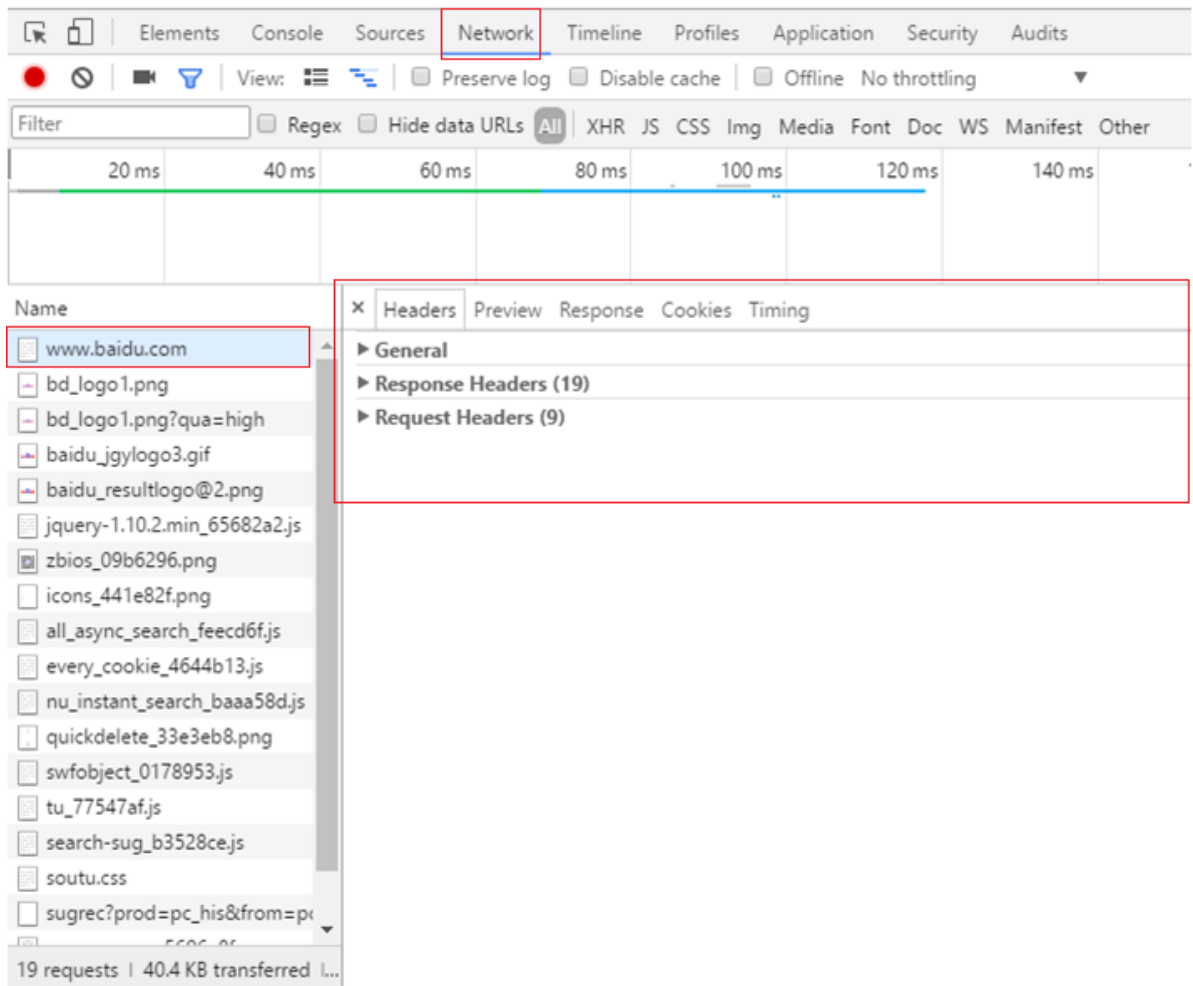


协议 域名 端口号 请求资源路径 (URI统一资源标识符)

2.3抓包工具的使用【应用】

- 使用步骤

1. 在谷歌浏览器网页中按F12 或者网页空白处右键,点击检查,可以调出工具
2. 点击network,进入到查看网络相关信息界面
3. 这时在浏览器中发起请求,进行访问,工具中就会显示出请求和响应相关的信息



2.4请求信息【理解】

- 组成
 - 请求行
 - 请求头
 - 请求空行
 - 请求体

- 请求行
 - 格式



- 请求方式
GET,POST,HEAD,PUT,DELETE,CONNECT,OPTIONS,TRACE,PATCH
其中用的比较多的是GET和POST
- URI
请求资源路径,统一资源标识符



- 协议版本
 - HTTP1.0: 每次请求和响应都需要建立一个单独的连接
 - HTTP1.1: 支持长连接
- 请求头
 - 格式

请求头名称	冒号	空格	请求头值	回车符	换行符
-------	----	----	------	-----	-----
 - 请求头名称
 - Host: 用来指定请求的服务端地址
 - Connection: 取值为keep-alive表示需要持久连接
 - User-Agent: 客户端的信息
 - Accept: 指定客户端能够接收的内容类型
 - Accept-Encoding: 指定浏览器可以支持的服务器返回内容压缩编码类型
 - Accept-Language: 浏览器可接受的语言

请求头

```
▶ General
▶ Response Headers (16)
▼ Request Headers view parsed
GET /index.html HTTP/1.1
Host: www.baidu.com
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
```

- 小结



2.5响应信息【理解】

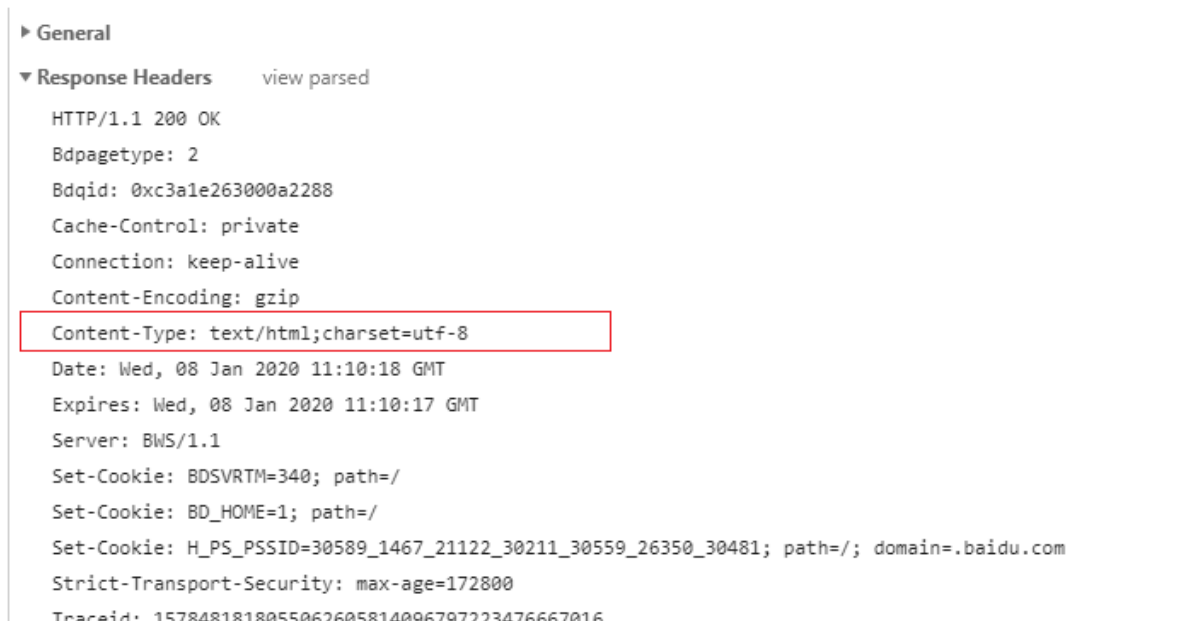
- 组成
 - 响应行
 - 响应头
 - 响应空行
 - 响应体

- 响应行
 - 格式



- 协议版本
 - HTTP1.0: 每次请求和响应都需要建立一个单独的连接
 - HTTP1.1: 支持长连接
- 响应状态码
 - 1xx: 指示信息(表示请求已接收, 继续处理)
 - 2xx: 成功(表示请求已被成功接收、理解、接受)
 - 3xx: 请求重定向(要完成请求必须进行更进一步的操作)
 - 4xx: 客户端错误(请求有语法错误或请求无法实现)
 - 5xx: 服务器端错误(服务器未能实现合法的请求)
- 状态信息
 - 200 ok
 - 404 Not Found
 - 500 Internal Server Error
- 响应头
 - 响应头名称
 - Content-Type: 告诉客户端实际返回内容的网络媒体类型(互联网媒体类型,也叫做MIME类型)
 - 响应头值

- text/html ----> 文本类型
- image/png ----> png格式文件
- image/jpeg ----> jpg格式文件



- 小结



3.HTTP服务器

3.1需求【理解】

- 编写服务器端代码,实现可以解析浏览器的请求,给浏览器响应数据

3.2环境搭建【理解】

- 实现步骤
 - 编写HttpServer类,实现可以接收浏览器发出的请求
 - 其中获取连接的代码可以单独抽取到一个类中
- 代码实现

// 服务端代码

```

public class HttpServer {
    public static void main(String[] args) throws IOException {
        //1.打开服务端通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        //2.让这个通道绑定一个端口
        serverSocketChannel.bind(new InetSocketAddress(10000));
        //3.设置通道为非阻塞
        serverSocketChannel.configureBlocking(false);
        //4.打开一个选择器
        Selector selector = Selector.open();

        //5.绑定选择器和服务端通道
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        while(true){
            //6.选择器会监视通道的状态.
            int count = selector.select();
            if(count != 0){
                //7.会遍历所有的服务端通道.看谁准备好了,谁准备好了,就让谁去连接.
                //获取所有服务端通道的令牌,并将它们都放到一个集合中,将集合返回.
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while(iterator.hasNext()){
                    //selectionKey 依次表示每一个服务端通道的令牌
                    SelectionKey selectionKey = iterator.next();
                    if(selectionKey.isAcceptable()){
                        //获取连接
                        AcceptHandler acceptHandler = new AcceptHandler();
                        acceptHandler.connSocketChannel(selectionKey);
                    }else if(selectionKey.isReadable()){
                        //任务处理完毕以后,将SelectionKey从集合中移除
                        iterator.remove();
                    }
                }
            }
        }
    }
}

// 将获取连接的代码抽取到这个类中
public class AcceptHandler {

    public SocketChannel connSocketChannel(SelectionKey selectionKey){
        try {
            //获取到已经就绪的服务端通道
            ServerSocketChannel ssc = (ServerSocketChannel) selectionKey.channel();
            SocketChannel socketChannel = ssc.accept();
            //设置为非阻塞状态
            socketChannel.configureBlocking(false);
            //把socketChannel注册到选择器上
            socketChannel.register(selectionKey.selector(), SelectionKey.OP_READ);
            return socketChannel;
        } catch (IOException e) {

```

```

        e.printStackTrace();
    }
    return null;
}
}

```

3.3获取请求信息并解析【理解】

- 实现步骤
 - 将请求信息封装到HttpRequest类中
 - 在类中定义方法,实现获取请求信息并解析
- 代码实现

```

/**
 * 用来封装请求数据的类
 */
public class HttpRequest {
    private String method; //请求方式
    private String requestURI; //请求的uri
    private String version; //http的协议版本

    private HashMap<String,String> hm = new HashMap<>(); //所有的请求头

    //parse --- 获取请求数据 并解析
    public void parse(SelectionKey selectionKey){
        try {
            SocketChannel socketChannel = (SocketChannel) selectionKey.channel();

            StringBuilder sb = new StringBuilder();
            //创建一个缓冲区
            ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
            int len;
            //循环读取
            while((len = socketChannel.read(byteBuffer)) > 0){
                byteBuffer.flip();
                sb.append(new String(byteBuffer.array(),0,len));
                //System.out.println(new String(byteBuffer.array(),0,len));
                byteBuffer.clear();
            }
            //System.out.println(sb);
            parseHttpRequest(sb);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //解析http请求协议中的数据
    private void parseHttpRequest(StringBuilder sb) {
        //1. 需要把StringBuilder先变成一个字符串
        String httpRequestStr = sb.toString();
        //2. 获取每一行数据
    }
}

```

```

String[] split = httpRequestStr.split("\r\n");
//3.获取请求行
String httpRequestLine = split[0]; //GET / HTTP/1.1
//4.按照空格进行切割,得到请求行中的三部分
String[] httpRequestInfo = httpRequestLine.split(" ");
this.method = httpRequestInfo[0];
this.requestURI = httpRequestInfo[1];
this.version = httpRequestInfo[2];
//5.操作每一个请求头
for (int i = 1; i < split.length; i++) {
    String httpRequestHeaderInfo = split[i]; //Host: 127.0.0.1:10000
    String[] httpRequestHeaderInfoArr = httpRequestHeaderInfo.split(": ");
    hm.put(httpRequestHeaderInfoArr[0], httpRequestHeaderInfoArr[1]);
}

}

public String getMethod() {
    return method;
}

public void setMethod(String method) {
    this.method = method;
}

public String getRequestURI() {
    return requestURI;
}

public void setRequestURI(String requestURI) {
    this.requestURI = requestURI;
}

public String getVersion() {
    return version;
}

public void setVersion(String version) {
    this.version = version;
}

public HashMap<String, String> getHm() {
    return hm;
}

public void setHm(HashMap<String, String> hm) {
    this.hm = hm;
}

@Override
public String toString() {
    return "HttpRequest{" +

        "method='" + method + '\'' +

```



```

        ", requestURI='" + requestURI + '\'' +
        ", version='" + version + '\'' +
        ", hm='" + hm +
        '}'';
    }
}

```

3.4给浏览器响应数据【理解】

- 实现步骤
 - 将响应信息封装HttpResponse类中
 - 定义方法,封装响应信息,给浏览器响应数据
- 代码实现

```

public class HttpResponse {
    private String version; //协议版本
    private String status; //响应状态码
    private String desc; //状态码的描述信息

    //响应头数据
    private HashMap<String, String> hm = new HashMap<>();

    private HttpRequest httpRequest; //我们后面要根据请求的数据,来进行一些判断

    //给浏览器响应数据的方法
    public void sendStaticResource(SelectionKey selectionKey) {
        //1.给响应行赋值
        this.version = "HTTP/1.1";
        this.status = "200";
        this.desc = "ok";
        //2.将响应行拼接成一个单独的字符串 // HTTP/1.1 200 ok
        String responseLine = this.version + " " + this.status + " " + this.desc + "\r\n";

        //3.给响应头赋值
        hm.put("Content-Type", "text/html;charset=UTF-8");

        //4.将所有的响应头拼接成一个单独的字符串
        StringBuilder sb = new StringBuilder();
        Set<Map.Entry<String, String>> entries = hm.entrySet();
        for (Map.Entry<String, String> entry : entries) {
            sb.append(entry.getKey()).append(": ").append(entry.getValue()).append("\r\n");
        }

        //5.响应空行
        String emptyLine = "\r\n";

        //6.响应行,响应头,响应空行拼接成一个大字符串
        String responseLineStr = responseLine + sb.toString() + emptyLine;

        try {
            //7.将上面三个写给浏览器
            SocketChannel socketChannel = (SocketChannel) selectionKey.channel();

```

```

        ByteBuffer byteBuffer1 = ByteBuffer.wrap(responseLineStr.getBytes());
        socketChannel.write(byteBuffer1);

        //8.单独操作响应体
        //因为在以后响应体不一定是一个字符串
        //有可能是一个文件,所以单独操作
        String s = "哎哟,妈呀,终于写完了.";
        ByteBuffer byteBuffer2 = ByteBuffer.wrap(s.getBytes());
        socketChannel.write(byteBuffer2);

        //9.释放资源
        socketChannel.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public String getVersion() {
    return version;
}

public void setVersion(String version) {
    this.version = version;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public String getDesc() {
    return desc;
}

public void setDesc(String desc) {
    this.desc = desc;
}

public HashMap<String, String> getHm() {
    return hm;
}

public void setHm(HashMap<String, String> hm) {
    this.hm = hm;
}

public HttpRequest getHttpRequest() {
    return httpRequest;
}

```

```

public void setHttpRequest(HttpRequest httpRequest) {
    this.httpRequest = httpRequest;
}

@Override
public String toString() {
    return "HttpResponse{" +
        "version='" + version + '\'' +
        ", status='" + status + '\'' +
        ", desc='" + desc + '\'' +
        ", hm=" + hm +
        ", httpRequest=" + httpRequest +
        '}';
}
}

```

3.5代码优化【理解】

- 实现步骤
 - 根据请求资源路径不同,响应不同的数据
 - 服务端健壮性处理
 - 访问不存在的资源处理
- 代码实现

```

/**
 * 接收连接的任务处理类
 */
public class AcceptHandler {

    public SocketChannel connSocketChannel(SelectionKey selectionKey){
        try {
            //获取到已经就绪的服务端通道
            ServerSocketChannel ssc = (ServerSocketChannel) selectionKey.channel();
            SocketChannel socketChannel = ssc.accept();
            //设置为非阻塞状态
            socketChannel.configureBlocking(false);
            //把socketChannel注册到选择器上
            socketChannel.register(selectionKey.selector(), SelectionKey.OP_READ);
            return socketChannel;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}

/**
 * 接收客户端请求的类
 */
public class HttpServer {
    public static void main(String[] args) throws IOException {

        //1.打开服务端通道
    }
}

```



```

*/
public class HttpRequest {
    private String method; //请求方式
    private String requestURI; //请求的uri
    private String version; //http的协议版本

    private HashMap<String,String> hm = new HashMap<>(); //所有的请求头

    //parse --- 获取请求数据 并解析
    public void parse(SelectionKey selectionKey){
        try {
            SocketChannel socketChannel = (SocketChannel) selectionKey.channel();

            StringBuilder sb = new StringBuilder();
            //创建一个缓冲区
            ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
            int len;
            //循环读取
            while((len = socketChannel.read(byteBuffer)) > 0){
                byteBuffer.flip();
                sb.append(new String(byteBuffer.array(),0,len));
                //System.out.println(new String(byteBuffer.array(),0,len));
                byteBuffer.clear();
            }
            //System.out.println(sb);
            parseHttpRequest(sb);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //解析http请求协议中的数据
    private void parseHttpRequest(StringBuilder sb) {
        //1.需要把StringBuilder先变成一个字符串
        String httpRequestStr = sb.toString();
        if(!(httpRequestStr == null || "".equals(httpRequestStr))){
            //2.获取每一行数据
            String[] split = httpRequestStr.split("\r\n");
            //3.获取请求行
            String httpRequestLine = split[0]; //GET / HTTP/1.1
            //4.按照空格进行切割,得到请求行中的三部分
            String[] httpRequestInfo = httpRequestLine.split(" ");
            this.method = httpRequestInfo[0];
            this.requestURI = httpRequestInfo[1];
            this.version = httpRequestInfo[2];
            //5.操作每一个请求头
            for (int i = 1; i < split.length; i++) {
                String httpRequestHeaderInfo = split[i]; //Host: 127.0.0.1:10000
                String[] httpRequestHeaderInfoArr = httpRequestHeaderInfo.split(": ");
                hm.put(httpRequestHeaderInfoArr[0],httpRequestHeaderInfoArr[1]);
            }
        }
    }
}

```

```

    }

    public String getMethod() {
        return method;
    }

    public void setMethod(String method) {
        this.method = method;
    }

    public String getRequestURI() {
        return requestURI;
    }

    public void setRequestURI(String requestURI) {
        this.requestURI = requestURI;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public HashMap<String, String> getHm() {
        return hm;
    }

    public void setHm(HashMap<String, String> hm) {
        this.hm = hm;
    }

    @Override
    public String toString() {
        return "HttpRequest{" +
            "method='" + method + '\'' +
            ", requestURI='" + requestURI + '\'' +
            ", version='" + version + '\'' +
            ", hm=" + hm +
            '}';
    }
}

/**
 * 用来封装响应数据的类
 */
public class HttpResponse {
    private String version; //协议版本
    private String status; //响应状态码
    private String desc; //状态码的描述信息

    //响应头数据

```

```

private HashMap<String, String> hm = new HashMap<>();

private HttpRequest httpRequest; //我们后面要根据请求的数据,来进行一些判断

//给浏览器响应数据的方法
public void sendStaticResource(SelectionKey selectionKey) {
    //1.给响应行赋值
    this.version = "HTTP/1.1";
    this.status = "200";
    this.desc = "ok";

    //3.给响应头赋值
    //先获取浏览器请求的URI
    String requestURI = this.getHttpRequest().getRequestURI();
    if(requestURI != null){

        File file = new File(WEB_APP_PATH + requestURI);
        //判断这个路径是否存在
        if(!file.exists()){
            this.status = "404";
            this.desc = "NOT FOUND";
        }

        if("200".equals(this.status)){
            if("/".equals(requestURI)){
                hm.put("Content-Type", "text/html;charset=UTF-8");
            }else if("/favicon.ico".equals(requestURI)){
                hm.put("Content-Type", "image/x-icon");
            }else if("/a.txt".equals(requestURI)){
                hm.put("Content-Type", "text/html;charset=UTF-8");
            }else if("/1.jpg".equals(requestURI)){
                hm.put("Content-Type", "image/jpeg");
            }else if("/1.png".equals(requestURI)){
                hm.put("Content-Type", "image/png");
            }
        }else{
            hm.put("Content-Type", "text/html;charset=UTF-8");
        }
    }

    //2.将响应行拼接成一个单独的字符串 // HTTP/1.1 200 ok
    String responseLine = this.version + " " + this.status + " " + this.desc + "\r\n";

    //4.把所有的响应头拼接成一个单独的字符串
    StringBuilder sb = new StringBuilder();
    Set<Map.Entry<String, String>> entries = hm.entrySet();
    for (Map.Entry<String, String> entry : entries) {
        sb.append(entry.getKey()).append(": ").append(entry.getValue()).append("\r\n");
    }

    //5.响应空行

    String emptyLine = "\r\n";

```

```

//6.响应行,响应头,响应空行拼接成一个大字符串
String responseLineStr = responseLine + sb.toString() + emptyLine;

try {
    //7.将上面三个写给浏览器
    SocketChannel socketChannel = (SocketChannel) selectionKey.channel();
    ByteBuffer byteBuffer1 = ByteBuffer.wrap(responseLineStr.getBytes());
    socketChannel.write(byteBuffer1);

    //8.单独操作响应体
    //因为在以后响应体不一定是一个字符串
    //有可能是一个文件,所以单独操作
    // String s = "哎哟,妈呀,终于写完了.";
    byte [] bytes = getContent();
    ByteBuffer byteBuffer2 = ByteBuffer.wrap(bytes);
    socketChannel.write(byteBuffer2);

    //9.释放资源
    socketChannel.close();
} catch (IOException e) {
    e.printStackTrace();
}

}

public static final String WEB_APP_PATH = "mynio\\webapp";
private byte[] getContent() {
    try {
        //1.获取浏览器请求的URI
        String requestURI = this.getHttpRequest().getRequestURI();
        if(requestURI != null){

            if("200".equals(this.status)){
                //2.判断一下请求的URI,根据不同的URI来响应不同的东西
                if("/".equals(requestURI)){
                    String s = "哎哟,妈呀,终于写完了.";
                    return s.getBytes();
                }else/* if("/favicon.ico".equals(requestURI))*/{
                    //获取一个ico文件
                    FileInputStream fis = new FileInputStream(WEB_APP_PATH +
requestURI);

                    //把ico文件变成一个字节数组返回
                    return IOUtils.toByteArray(fis);
                }
            }else{
                return "访问的资源不存在".getBytes();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return new byte[0];
}
}

```



```

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    public HashMap<String, String> getHm() {
        return hm;
    }

    public void setHm(HashMap<String, String> hm) {
        this.hm = hm;
    }

    public HttpRequest getHttpRequest() {
        return httpRequest;
    }

    public void setHttpRequest(HttpRequest httpRequest) {
        this.httpRequest = httpRequest;
    }

    @Override
    public String toString() {
        return "HttpResponse{" +
            "version='" + version + '\'' +
            ", status='" + status + '\'' +
            ", desc='" + desc + '\'' +
            ", hm=" + hm +
            ", httpRequest=" + httpRequest +
            '}';
    }
}

```

