**随堂测试③**

一．单选题（每空3分），共30分。在以下每小题给出的A 、B、C、D四个选项中，只有一个选项正确，请选出正确答案，并将答案填在题中括号内）。

1. 关于缓存及属性相关的描述，不正确的是（ C ）。
2. 如果缓存没超过过期时间，则无需与服务器交互，直接使用
3. Etag首部用于资源改变时唯一的标识符
4. no-cache表示不能缓存
5. Last-modified表示资源最后一次修改的日期
6. 关于socket.setOOBInline的用法描述，不正确的是（ B ）。
7. 在flush方法前，能优先发送数据
8. 只需客户端设置，服务器端不需要设置
9. OOB 是 Out-of-band的缩写，发送具有特殊意义的紧急数据
10. 每次只能发送一个字节数据
11. 关于保护通信相关的描述，不正确的是（ A ）。
12. 实际应用中，使用对称加密传输非对称加密的密钥
13. 对于对称加密，加密方和解密方使用相同的密钥
14. 对于非对称加密，加密方和解密方使用不同的密钥
15. 为了避免中间人攻击，公开密钥放在可信任的第三方认证机构
16. 方法（ C ）用于判断远端服务器是否已经断开连接最合适。
17. isClosed
18. isConnected
19. sendUrgentData
20. isInputStreamShutdown
21. 为了防止客户端不知服务器已经崩溃，客户端可以采用方法（ B ）在空闲时持续检查服务器的响应情况。
22. setTcpNoDelay
23. setKeepAlive
24. setSoTimeout
25. setSoLinger
26. Buffer的flip() 方法的作用是（ A ）。
27. 将限度limit设置为当前位置position，位置position设置为0；
28. 将限度limit设置为capacity，位置position设置为0；
29. 将限度limit设置为0，位置position设置为当前位置position
30. 将限度limit设置为0，位置position设置为capacity；
31. 关于服务器日志的描述，正确的是（ C ）。
32. 日志可有可无
33. 一般不用记录服务器错误
34. 系统上线后，不应输出调试信息
35. 尽可能多的记录信息
36. 关于非阻塞I/O的优缺点描述，不正确的是（ B ）。
37. 能使单线程处理多个连接
38. 性能超过多线程+阻塞IO
39. 适用于很多连接，但每个连接不活跃的情况
40. 非阻塞IO处理连接是异步的
41. 对于安全Socket加密算法SSL\_DH\_anon\_EXPORT\_WITH\_DES40\_CBC\_SHA，DES40表示（ C ）。
42. 密码协商的方法
43. 密钥交换算法
44. 加密算法
45. 校验和
46. 下面函数（ B ）的作用是将 position 与 limit之间的数据复制到buffer的开始位置，并分别将limit –position，capacity赋值给position与limit。
47. reset
48. compact
49. rewind
50. flip

二．判断题 （每小题2分，共10分，请给正确的打“√”，错误的打“×”，并填写题中括号内）。

1. 使用Java.net.Socket可以直接创建一个安全socket对象（ F ）
2. 默认情况下，SocketChannel对象处于阻塞模式（ T ）
3. 在SSL协议中， 当一个通信端无须向对方证实自己的身份，就称它处于客户模式（ T ）
4. UDPSocket和TCPsocket的SO\_REUSEADDR选项作用相同（ F ）
5. 下列两段代码是等价的 （ F ）

//第一段

socket.shutdownInput()

socket.shutdownOutput()

//第二段

socket.close

三．综合题（每小题30分，共60分）

1. 请简述什么是中间人攻击，如何有效解决？

从公有密钥和私有密钥的角度来看，中间人攻击是指攻击者通过欺骗双方，分别与双方建立起两个虚假的密钥对，使得攻击者能够在通信过程中拦截、篡改或伪造数据。

中间人攻击的过程如下：

攻击者与通信的双方分别建立联系，并向每个参与者发送一个虚假的公有密钥。

通信的双方会使用这些虚假的公有密钥进行加密和解密操作，而攻击者则能够获得解密后的数据。

攻击者将数据篡改后再用虚假的公有密钥重新加密，然后分别发送给双方，使得双方误以为是对方发来的数据。

双方在收到攻击者篡改的数据后，使用虚假的私有密钥进行解密，无法察觉数据已经被篡改。

为了有效解决中间人攻击，可以采取以下措施：

数字证书：通信的双方可以使用数字证书来验证对方的身份。数字证书由可信的第三方机构颁发，包含了公有密钥和相关信息。双方可以使用数字证书验证对方的身份，并确保使用的公有密钥是合法的。

针对中间人攻击的协议：一些安全通信协议，例如SSL/TLS，采用了防止中间人攻击的机制。这些协议在握手过程中使用复杂的加密算法和密钥交换方式，确保双方能够安全地建立连接，并验证对方的身份。

增强密钥管理：确保生成、存储和交换密钥的过程是安全可靠的。密钥应该由受信任的实体生成，并妥善保管，以防止密钥被攻击者获取或篡改。

安全传输通道：使用安全的传输通道，如HTTPS，通过加密通信的方式，保护数据在传输过程中的机密性和完整性。

安全意识培训：教育用户如何辨别和防范中间人攻击。用户应该学会警惕浏览器的安全警告，验证证书的有效性，避免在不安全的网络环境下进行敏感信息的传输。

综上所述，通过使用数字证书、采用针对中间人攻击的协议、增强密钥管理、使用安全传输通道以及提高用户安全意识，可以有效解决中间人攻击问题。这些措施能够确保通信双方的身份验证和数据的机密性，减少中间人攻击的风险。

1. 使用Java编程实现web缓存可以加快客户端访问web的相应速度，需要使用系统级缓存。如果在你编程实现的HTTP客户端需具有缓存功能，你该如何去编程实现，需要使用哪些类，需要自定义哪些类，试给出编程实现的例子程序，例子程序可以直接给出源代码及说明，也可只给出详细类图，并指出类图中每一个类的作用。

要在Java中实现具有缓存功能的HTTP客户端，你可以使用以下类和自定义一些类来完成。

java.net.URL：用于表示要访问的URL。

java.net.HttpURLConnection：用于建立与服务器的HTTP连接，并发送HTTP请求。

java.io.InputStream：用于从服务器接收响应数据流。

java.io.OutputStream：用于将请求数据发送给服务器。

java.util.HashMap：用于实现缓存，将URL作为键，响应数据作为值。

自定义的类：

CacheManager：缓存管理类，用于实现缓存的初始化、添加、获取和删除等功能。

HttpRequestCache：自定义的HTTP请求缓存类，包含了请求URL和响应数据。

1. **import** java.io.\*;
2. **import** java.net.HttpURLConnection;
3. **import** java.net.URL;
4. **import** java.util.HashMap;
6. **public** **class** CachingHttpClient {
7. **private** HashMap<String, **byte**[]> cache;
9. **public** CachingHttpClient() {
10. cache = **new** HashMap<>();
11. }
13. **public** **byte**[] get(String urlStr) **throws** IOException {
14. **if** (cache.containsKey(urlStr)) {
15. System.out.println("Returning response from cache...");
16. **return** cache.get(urlStr);
17. }
19. URL url = **new** URL(urlStr);
20. HttpURLConnection connection = (HttpURLConnection) url.openConnection();
21. connection.setRequestMethod("GET");
23. **int** responseCode = connection.getResponseCode();
24. **if** (responseCode == HttpURLConnection.HTTP\_OK) {
25. InputStream inputStream = connection.getInputStream();
26. ByteArrayOutputStream outputStream = **new** ByteArrayOutputStream();
27. **byte**[] buffer = **new** **byte**[1024];
28. **int** bytesRead;
29. **while** ((bytesRead = inputStream.read(buffer)) != -1) {
30. outputStream.write(buffer, 0, bytesRead);
31. }
32. **byte**[] responseBytes = outputStream.toByteArray();
33. cache.put(urlStr, responseBytes);  // 添加到缓存
34. System.out.println("Response added to cache.");
35. **return** responseBytes;
36. }
38. **throw** **new** IOException("Failed to fetch data from server. Response code: " + responseCode);
39. }
41. **public** **static** **void** main(String[] args) {
42. CachingHttpClient client = **new** CachingHttpClient();
44. **try** {
45. **byte**[] response1 = client.get("http://example.com/page1");
46. // 此时，数据已从服务器获取并添加到缓存中
48. **byte**[] cachedResponse = client.get("http://example.com/page1");
49. // 直接从缓存中获取数据，而不需要再次访问服务器
51. **byte**[] response2 = client.get("http://example.com/page2");
52. // 访问另一个URL，数据将从服务器获取并添加到缓存中
54. **byte**[] cachedResponse2 = client.get("http://example.com/page2");
55. // 直接从缓存中获取数据
57. } **catch** (IOException e) {
58. e.printStackTrace();
59. }
60. }
61. }