ActionScript3.0 Socket编程

与Socket服务器建立连接

解决方法:

我们通过调用 Socket.connect( )或者 XMLSocket.connect( )方法并监听网络连接的事件消息.

讨论:

连接一台 Socket 服务器你需要确定两个信息,一个是 Socket 服务器的域名或者 IP 地址,另一个是服务器监听的端口号.

无论你使用的是 Socket 还是 XMLSocket 类的实例,连接请求都是完全的一样的,两个类都是使用一个名叫 connect()的方法,该方法有两个参数:

host :

该参数为字符串类型,可以是一个域名,例如"www.example.com",也可以是一个 IP 地址,例如"192.168.1.101".如果 Socket 服务器与你该 Flash 影片发布的 Web 服务器是同一个,该参数为 Null.

port :

该参数为一个表示 Socket 服务器监听端口的 int 值.该值最小为 1024.除非在服务器中有一个 policy 文件,用于指定允许端口号小于 1024.

因为 Flash Socket 编程是一个异步的过程,connect()方法不会等到一个连接完成后再执行下一行代码的执行.如果你想在一个连接完全执行完之前与一个 Socket 完全绑定,那么你将会得到一个意想不到的结果,并且你当前的代码将不能工作.

在尝试一个新的 Socket 连接的时候我们最好先添加一个连接事件监听器.当一个连接建立成功,Socket 或者 XMLSocket 会发出一个连接事件,这就可以让你知道交互已经准备好了.

下面举了一个 Socket 实例与本地 Socket 服务器的 2900 端口建立连接的例子:

```
package {
   import flash.display.Sprite;
   import flash.events.*;
   import flash.net.Socket;

   public class SocketExample extends Sprite {

      private var socket:Socket;

      public function SocketExample(    ) {
         socket = new Socket(    );
```

```
            // Add an event listener to be notified when the connection
            // is made
            socket.addEventListener( Event.CONNECT, onConnect );

            // Connect to the server
            socket.connect( "localhost", 2900 );
        }

        private function onConnect( event:Event ):void {
            trace( "The socket is now connected..." );
        }

    }
}
```

如果你想通过 XMLSocket 与服务器建立连接代码也是基本一样的.首先你创建了一个连接事件监听器,然后调用 connect()方法.所不同的是 Socket 实例改为了 XMLSocket:

```
package {
    import flash.display.Sprite;
    import flash.events.*;
    import flash.net.XMLSocket;

    public class SocketExample extends Sprite {

        private var socket:XMLSocket;

        public function SocketExample(    ) {
            socket = new XMLSocket(    );

            // Add an event listener to be notified when the connection is made
            socket.addEventListener( Event.CONNECT, onConnect );

            // Connect to the server
            socket.connect( "localhost", 2900 );
        }

        private function onConnect( event:Event ):void {
            trace( "The xml socket is now connected..." );
        }

    }
}
```

如果连接失败,可那是下面两种原因的一种:一种是连接立即失败和运行时错误,另一种是如果无法完成连接从而产生一个 ioError 或者 securityError 事件.关于错误事件处理信息的描述,我们打算改日讨论.

请牢记,当与一个主机建立一个 Socket 连接时,Flash Player 要遵守如下安全沙箱规则.

1.Flash 的.swf 文件和主机必须严格的在同一个域名,只有这样才可以成功建立连接.

2.一个从网上发布的.swf 文件是不可以访问本地服务器的.

3.本地未通过认证的.swf 文件是不可以访问任何网络资源的.

4.你想跨域访问或者连接低于 1024 的端口,必须使用一个跨域策略文件.

如果尝试连接未认证的域或者低端口服务,这样就违反了安全沙箱策略,同时会产生一个 securityError 事件.这些情况都可以通过使用一个跨域策略文件解决.无论是 Socket 对象还是 XMLSocket 对象的策略文件,都必须在连接之前通过使用 flash.system.Security.loadPolicyFile() 方法载入策略文件.具体如下:

Security.loadPolicyFile("http://www.rightactionscript.com/crossdomain.xml");

获得的改策略文件不仅定义了允许的域名,还定义了端口号.如果你不设置端口号,那么 Flash
Player 默认为 80 端口(HTTP 协议默认端口).在<allow-access-from>标签中可以使用逗号隔开设置多个端口号.下面这个例子就是允许访问 80 和 110 端口.

<?xmlversion="1.0"?><!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd"><cross-domain-policy>
<allow-access-from domain="*" to-ports="80,110" /></cross-domain-policy>

从Socket服务器读数据

解决方法:

对于 Socket 实例,先收到 socketData 事件,然后调用如下两个方法的一个,比如,readByte()或者 readInt(),在事件控制器中确定不会去读过去的 bytesAvailable.

对于 XMLSocket 实例,先收到 data 事件,然后解析从事件控制器内部装载的 XML 数据.

讨论:

从一个 socket 连接接收的数据依赖于你使用的 Socket 的类型.socket 和 XMLSocket 都可以从服务器接受到数据,但是它们处于不同重量级的技术.让我们在讨论 XMLSocket 之前先关注下 Socket 类.

我都知道 socket 在 Flash 中是一个异步的行为.因此,它就不能简单的创建一个 Socket 连接,然后就立刻尝试去读取数据.read 方法不能等到从服务器传过来数据之后在返回.换句话说,你只能在客户端从服务器载入所有数据之后才可以读取数据.在数据可用之前读数据会产生一个错误.

通过 socketData 事件广播到 Socket 实例,这样我们就可以知道什么时候数

据可以被读取.那么我们要为 socketData 事件添加一个事件监听器,任何时候只要有新的数据从一个 socket 服务器发送过来,都会触发事件控制器.在事件处理器的内部我们写入我们要执行的代码去读取和处理收到的数据.

从一个前端服务器读取数据,Socket 类为我们提供了许多不同的方法,这些方法依赖于你所读得数据类型.例如,你可以通过 readByte()方法读一个 byte 数据,或者通过一个使用 readUnsignedInt()方法去读一个无符号整数.下面这个表列出来能够从服务器读取的数据类型,返回值,和 read 方法每次读入的字节数.

Table:Socket read methods for various datatypes

方法:返回值类型	描述	字节数
readBoolean( ):Boolean	从 Socket 读取一个 Boolean 值.	1
readByte( ):int	从 Socket 读取一个 byte 值.	1
readDouble( ):Number	从 Socket 读取一个 IEEE 754 双精度浮点数.	8
readFloat( ):Number	从 Socket 读取一个 IEEE 754 单精度浮点数.	4
readInt( ):int	从 Socket 读取一个有符号 32-bit 整数值.	4
readObject( ):*	从 Socket 读取一个 AMF-encoded 对象.	n
readShort( ):int	从 Socket 读取一个有符号 16-bit 整数值.	2
readUnsignedByte( ):uint	从 Socket 读取一个无符号字节.	1
readUnsignedInt( ):uint	从 Socket 读取一个无符号 32-bit 整数	4
readUnsignedShort( ):uint	从 Socket 读取一个无符号 16-bit 整数.	2
readUTF( ):String	从 Socket 读取一个一个 UTF8 字符串.	n

有两个额外的方法没有在上面这个表中描述.它们分别是 readBytes()和 readUTFBytes().readBytes()方法只可以让 socket 读数据但不能返回一个值,并且该方法需要 3 个参数:

bytes:

一个 flash.util.ByteArray 实例读取从 socket 中收到的数据.

offset:
一个uint值,指定从什么位置开始读取socket中收到数据的偏移量.默认值为0.
length:
一个uint值,用于指定读取bytes的数量.默认值为0,意思就是说将所有的可用的数据都放入ByteArray中.

另一个readUTFBytes()方法,只需要一个长度参数用于指定UTF-8字节的读入数量,并且该方法会将所有读入的字节码转换成为字符串类型.

注意:在从一个Socket读数据之前,首先要判断bytesAvailable的属性.如果你不知道要读入的数据类型是什么就去读数据的话,将会产生一个错误(flash.errors.EOFError).

下面的例子代码连接了一个 socket 服务器,读取并显示每次从服务器发来的数据.

```
package {
  import flash.display.Sprite;
  import flash.events.ProgressEvent;
  import flash.net.Socket;

  public class SocketExample extends Sprite {

    private var socket:Socket;

    public function SocketExample(  ) {
      socket = new Socket(  );

      // Listen for when data is received from the socket server
            socket.addEventListener(       ProgressEvent.SOCKET_DATA,
onSocketData );

      // Connect to the server
      socket.connect( "localhost", 2900 );
    }

    private function onSocketData( event:ProgressEvent ):void {
      trace( "Socket received " + socket.bytesAvailable + " byte(s) of
data:" );

      // Loop over all of the received data, and only read a byte if there
      // is one available
      while ( socket.bytesAvailable ) {
```

```
            // Read a byte from the socket and display it
            var data:int = socket.readByte(  );
            trace( data );
        }
    }
  }
}
```

在上面的这个例子中,如果一个 socket 服务器发送回一个消息(例如"hello"),当一个客户段连入服务器就会返回并输出下面类似的文字:

```
Socket received 5 byte(s) of data:
72
101
108
108
111
```

注意:一旦数据从 socket 读出,它就不能再次被读.例如,读一个字节之后,这个字节就不能再"放回来",只能读后边的字节.

当收到的数据为 ASCII 编码,你可以通过 readUTFBytes()方法重新构建一个字符串.readUTFBytes()方法需要知道多少个字节需要转换为字符串.你可以使用 bytesAvailable 去读所有的字节数据:

```
var string:String = socket.readUTFBytes(socket.bytesAvailable);
```

XMLSocket 类的动作和 Socket 类相比在从服务器接受数据的风格相似.两者都是通过事件监听器来监听数据接收通知的,这主要取决于 Flash 异步的 Socket 实现.然而,在处理实际数据的时候有很大的不同.

有个 XMLSocket 实例在从服务器下载完数据后分发数据事件.通过 flash.events.DataEvent.DATA 常量定义的数据事件包含一个 data 属性,该属性包含了从服务器收到的信息.

注意:使用 XMLSocket 从服务器返回的数据总是认为是一个字符串类型的数据.这样不用为任何数据类型的数据指定读取方法.

这些从服务器返回的数据是没有经过任何处理的原始数据.因此,你不能通过 XMLSocket 连接立即使用 XML,你发送和接收的都是纯字符串数据.如果你期望 XML,在你处理数据之前,你必须首先将这些数据转换为一个 XML 的实例.

下面的这段代码在初始化的时候通过 XMLSocket 连接到了本地服务器的 2900 端口.在连接成功之后,一个<test>消息会发送到服务器.onData 事件监听者控制从服务器返回的响应.在本例中返回字符串<response><test

success='true'/></response>.你可以通过事件的 data 属性发现为字符串数据,然后 XML 类的构造函数将字符串转换成为了 XML 实例.最后,通过使用 E4X 语法的 XML 实例的一部分信息.(关于通过使用 E4X 处理 XML 的更多详细信息,我们需要另外讨论.)

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.DataEvent;
  import flash.net.XMLSocket;

  public class SocketExample extends Sprite {

    private var xmlSocket:XMLSocket;

    public function SocketExample(  ) {
      xmlSocket = new XMLSocket(  );

      // Connect listener to send a message to the server
      // after we make a successful connection
      xmlSocket.addEventListener( Event.CONNECT, onConnect );

      // Listen for when data is received from the socket server
      xmlSocket.addEventListener( DataEvent.DATA, onData );

      // Connect to the server
      xmlSocket.connect( "localhost", 2900 );
    }

    private function onConnect( event:Event ):void {
      xmlSocket.send( "<test/>" );
    }

    private function onData( event:DataEvent ):void {
      // The raw string returned from the server.
      // It might look something like this:
      // <response><test success='true'/></response>
      trace( event.data );

      // Convert the string into XML
      var response:XML = new XML( event.data );

      // Using E4X, access the success attribute of the "test"
      // element node in the response.
```

```
        // Output: true
        trace( response.test.@success );
      }
    }
}
```

注意:在 data 事件分发数据之前,XMLSocket 实例必须从服务器收到一个表示为空的 byte('\\0').也就是说,从服务器仅仅只发送所需要的字符串是不够的,必须在结尾处加入一个表示为空的 byte.

同Socket服务器进行握手,并确定收到了什么样的数据和如何处理这些数据

解决方法:

创建不同的常量来声明协议的状态.使用这些常量将指定的处理函数映射到相应的状态.在一个 socketData 事件控制器中,通过状态映射调用这些函数的.

讨论:

建立 Socket 连接通常要处理握手这个环节.尤其是在服务器初始化需要向客户端发送数据.然后客户端通过一种特殊的方式相应这些数据,接着服务器因此再次响应.整个处理过程直到握手完成并且建立起一个"正常的"连接为止.

处理服务器的不同响应是非难的,主要的原因是 socketData 事件控制器不能保存上下文的顺序.也就是说,服务器的响应不会告诉你"为什么"响应,也不告诉你这些响应数据被那个处理程序来处理.要想知道如何处理这些从服务器返回的响应不能从响应的本身来获得,尤其在响应变化的时候.或许一个响应返回了两个字节码,另一个返回了一个整数值还跟了一个双精度浮点数.这样看来让响应本身处理自己是一大难题.

我们通过创建一个状态量来标注不同的上下文,服务器通过这些上下文将数据发送到客户端.与这些状态量都有一个相关联的函数来处理该数据,这样你就可以很轻松的按照当前的协议状态去调用正确的处理函数.

当你要与一个 Socket 服务器建立连接需要考虑如下几个步骤:

1.当与服务器连接的时候,服务器立刻返回一个标志服务器可以支持的最高协议版本号的整数值.
2.客户端在响应的时候会返回一个实际使用协议的版本号.
3.服务器返回一个 8byte 的鉴定码.
4.然后客户端将这鉴定码返回到服务器.
5.如果客户端的响应不是服务器端所期望的,或者,就在这个时候该协议变成了一个常规操作模式,于是握手结束.

实际上在第四步可以在鉴定码中包含更多的安全响应.你可以通过发送各种加密方法的密匙来代替逐个发送的鉴定码.这通常使用在客户端向用户索要密码

的时候,然后密码成为了加密过的 8byte 鉴定码.该加密过的鉴定码接着返回到服务器.如果响应的鉴定码匙服务器所期望的,客户端就知道该密码是正确的,然后同意建立连接.

实现握手框架,你首先要为处理从服务器返回的不同类型的数据分别创建常量.首先,你要从步骤 1 确定版本号.然后从步骤 3 收取鉴定码.最后就是步骤 5 的常规操作模式.我们可以声明

如下常量:

```
public const DETERMINE_VERSION:int = 0;
public const RECEIVE_CHALLENGE:int = 1;
public const NORMAL:int = 2;
```

常量的值并不重要,重要的是这些值要是不同的值,两两之间不能有相同的整数值.

下一个步骤我们就要为不同的数据创建不同处理函数了.创建的这三个函数分别被命名为 readVersion( ), readChallenge( ) 和 readNormalProtocol( ).创建完这三个函数后,我们就必须将这三个函数分别映射到前面不同状态常量,从而分别处理在该状态中收到的数据.代码如下:

```
stateMap = new Object(  );
stateMap[ DETERMINE_VERSION ] = readVersion;
stateMap[ RECEIVE_CHALLENGE ] = readChallenge;
stateMap[ NORMAL            ] = readNormalProtocol;
```

最后一步是编写 socketData 事件处理控制器,只有通过这样的方式,建立在当前协议状态之上的正确的处理函数才可以被调用.首先需要创建一个 currentState 的 int 变量.然后使用 stateMap 去查询与 currentState 相关联的函数,这样处理函数就可以被正确调用了.

```
var processFunc:Function = stateMap[ currentState ];
processFunc(  ); // Invoke the appropriate processing function
```

下面是一点与薄记相关的处理程序.在你的代码中更新 currentState 从而确保当前协议的状态.

前面我们所探讨的握手步骤的完整的代码如下:

```
package {
  import flash.display.Sprite;
  import flash.events.ProgressEvent;
  import flash.net.Socket;
  import flash.utils.ByteArray;
```

```
public class SocketExample extends Sprite {

  // The state constants to describe the protocol
  public const DETERMINE_VERSION:int = 0;
  public const RECEIVE_CHALLENGE:int = 1;
  public const NORMAL:int = 2;

  // Maps a state to a processing function
  private var stateMap:Object;

  // Keeps track of the current protocol state
  private var currentState:int;

  private var socket:Socket;

  public function SocketExample( ) {
    // Initialzes the states map
    stateMap = new Object( );
    stateMap[ DETERMINE_VERSION ] = readVersion;
    stateMap[ RECEIVE_CHALLENGE ] = readChallenge;
    stateMap[ NORMAL            ] = readNormalProtocol;

    // Initialze the current state
    currentState = DETERMINE_VERSION;

    // Create and connect the socket
    socket = new Socket( );
          socket.addEventListener(       ProgressEvent.SOCKET_DATA,
onSocketData );
    socket.connect( "localhost", 2900 );
  }

  private function onSocketData( event:ProgressEvent ):void {
    // Look up the processing function based on the current state
    var processFunc:Function = stateMap[ currentState ];
    processFunc( );
  }

  private function readVersion( ):void {
    // Step 1 - read the version from the server
    var version:int = socket.readInt( );

    // Once the version is read, the next state is receiving
```

```
      // the challenge from the server
      currentState = RECEIVE_CHALLENGE;

      // Step 2 - write the version back to the server
      socket.writeInt( version );
      socket.flush(  );
    }

    private function readChallenge(  ):void {
      // Step 3 - read the 8 byte challenge into a byte array
      var bytes:ByteArray = new ByteArray(  );
      socket.readBytes( bytes, 0, 8 );

      // After the challenge is received, the next state is
      // the normal protocol operation
      currentState = NORMAL;

      // Step 4 - write the bytes back to the server
      socket.writeBytes( bytes );
      socket.flush(  );
    }

    private function readNormalProtocol(  ):void {
      // Step 5 - process the normal socket messages here now that
      // that handshaking process is complete
    }
  }
}
```

与Socket服务器断开,或者当服务器想与你断开的时候发消息给你

解决方法:

通过调用 Socket.close()或者 XMLSocket.close()方法显性的断开与服务器的连接.同时可以通过监听 close 事件获得服务器主动断开的消息.

讨论:

通常情况下我们需要对程序进行下清理工作.比如说,你创建了一个对象,当这个对象没有用的时候我们就要删除它.因此,无论我们什么时候连接一个 Socket 服务器,都要在我们完成了必要的任务之后显性的断开连接. 一直留着无用的 Socket 连接浪费网络资源,应该尽量避免这种情况.如果你没有断开一个连接,那么这个服务器会继续保持着这个无用的连接.这样一来就很快会超过了服务器最大 Socket 连接上线.

Socket 和 XMLSocket 对象断开连接的方法是一样的.你只需要调用 close()方法就可以了:

```
// Assume socket is a connected Socket instance
socket.close( );  // Disconnect from the server
```

同样的,XMLSocket 对象断开连接的方法一样:

```
// Assume xmlSocket is a connected XMLSocket instance
xmlSocket.close( );  // Disconnect from the server
```

close()方法用于通知服务器客户端想要断开连接.当服务器主动断开连接会发消息通知客户端.可以通过调用 addEventListener()方法注册一个 close 事件的一个监听器.Socket 和 XMLSocket 都是使用 Event.CLOSE 作为"连接断开"事件类型的;例如:

```
// Add an event listener to be notified when the server disconnects
// the client
socket.addEventListener( Event.CLOSE, onClose );
```

注意:调用 close()方法是不会触发 close 事件的,只用服务器主动发起断开才会触发.一旦一个 Socket 断开了,就无法读写数据了.如果你想要从新这个连接,你只能再建立个新的连接了.

处理使用Sockets时候引发的错误

解决方法:

使用 try/catch 处理 I/O 和 EOF(end of file)错误.

讨论:

Socket 和 XMLSocket 类对错误的处理很类似.不如,当调用 connect()方法的时候,在下面任何一个条件成立的情况下 Socket 和 XMLSocket 对象会抛出一个类型为 SecurityError 的错误.

* 该.swf 未通过本地安全认证.
* 端口号大于 655535.

当调用 XMLSocket 对象的 send()或者 Socket 对象的 flush()的时候,如果 socket 还没有连接这两个方法都会抛出一个类型为 IOError 的错误.尽管你可以将 send()或者 flush()方法放入 try/catch 结构块中,你也不能依赖于 try/catch 结构块作为你应用程序的逻辑.更好的办法是,在调用 send()或者 flush()方法之前使用一个 if 语句首先判断一下 Socket 对象的 connected 属性是否为 True.例如,下面的代码使用了 if 语句作为程序逻辑的一部分,当 Socket 对象当前不是连接状态就调用 connectToSocketServer()方法.但是我们依然需要将 flush()方法放到 try/catch 语句块中.通过使用 try/catch 语句块将 flush()方法抛出的错误写入到日志中:

```
if ( socket.connected ) {
    try {
        socket.flush(   );
    }
    catch( error:IOError ) {
        logInstance.write( "socket.flush error\n" + error );
    }
}
else {
    connectToSocketServer(   );
}
```

所有的 Socket 类的 read 方法都能够抛出 EOFError 和 IOError 类型的错误.当你试图读一个数据,但是没有任何可用数据将触发 EOF 错误.当你试图从一个已经关闭的 Socket 对象中对数据时将会抛出 I/O 错误.

除了 Socket 和 XMLSocket 类的方法能够抛出的错误以外,这些类的对象还会分发错误事件.有两种基本的错误事件类型,他们分别由 socketIOError 和 securityError 错误引起.IOError 事件为 IOErrorEvent 类型,当数据发送或接收失败触发该事件.SecurityError 事件是 SecurityErrorEvent 类型,当一个 Socket 尝试连接一个服务器,但由于服务器不在安全沙箱范围之内或者端口号小于 1024 的时候触发该错误事件.

注意:这两种安全策略引起的错误都可以通过跨域访问策略文件解决.