Mtime SimpleRemoteService 逻辑设计文档

版本记录

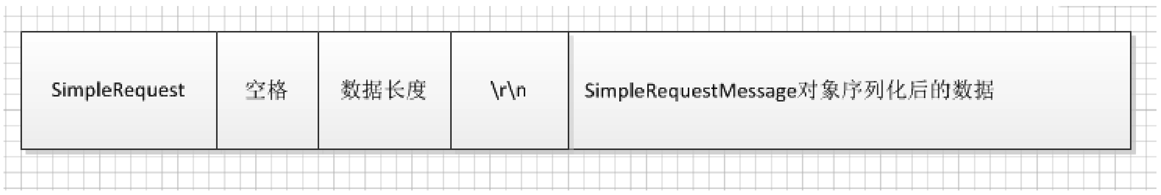
修改日期	修改人	修改版本	修改内容摘要

目　录

1 发送

简单说：客户端需要按照协议将所需内容发送到服务器端，SimpleRequestMessage 中封装了对应需要请求的服务名称，服务的方法名称，方法所需的参数及自定义的 Cookie 列表

1.1 协议



1.2 SimpleRequestMessage

```
message SimpleRequestCookie {
	required string Name = 1;
	required string Value = 2;
	required int64 ExpireTime = 3;
}
```

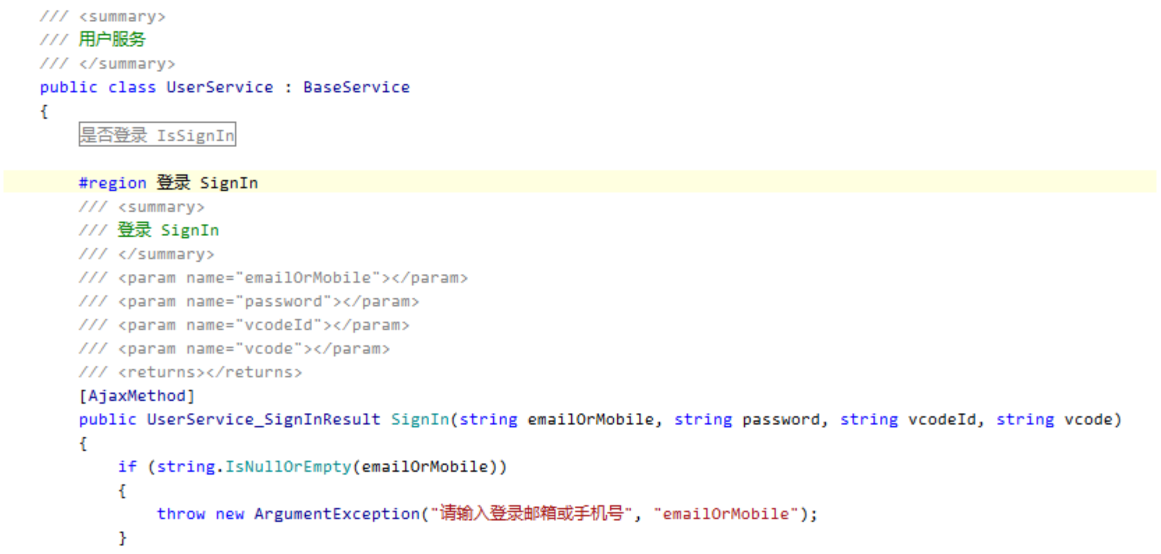
```
message SimpleRequestParameter {
	required int32 DataType = 1;
	required bytes Data = 2;
}

message SimpleRequestMessage {
	required string ClientId = 1;
	optional string UserToken = 2;
	required string ServiceName = 3;
	required string MethodName = 4;
	repeated SimpleRequestParameter Parameters = 5;
	repeated SimpleRequestCookie Cookies = 6;
}
```

1.3 例子

1.3.1 服务器端

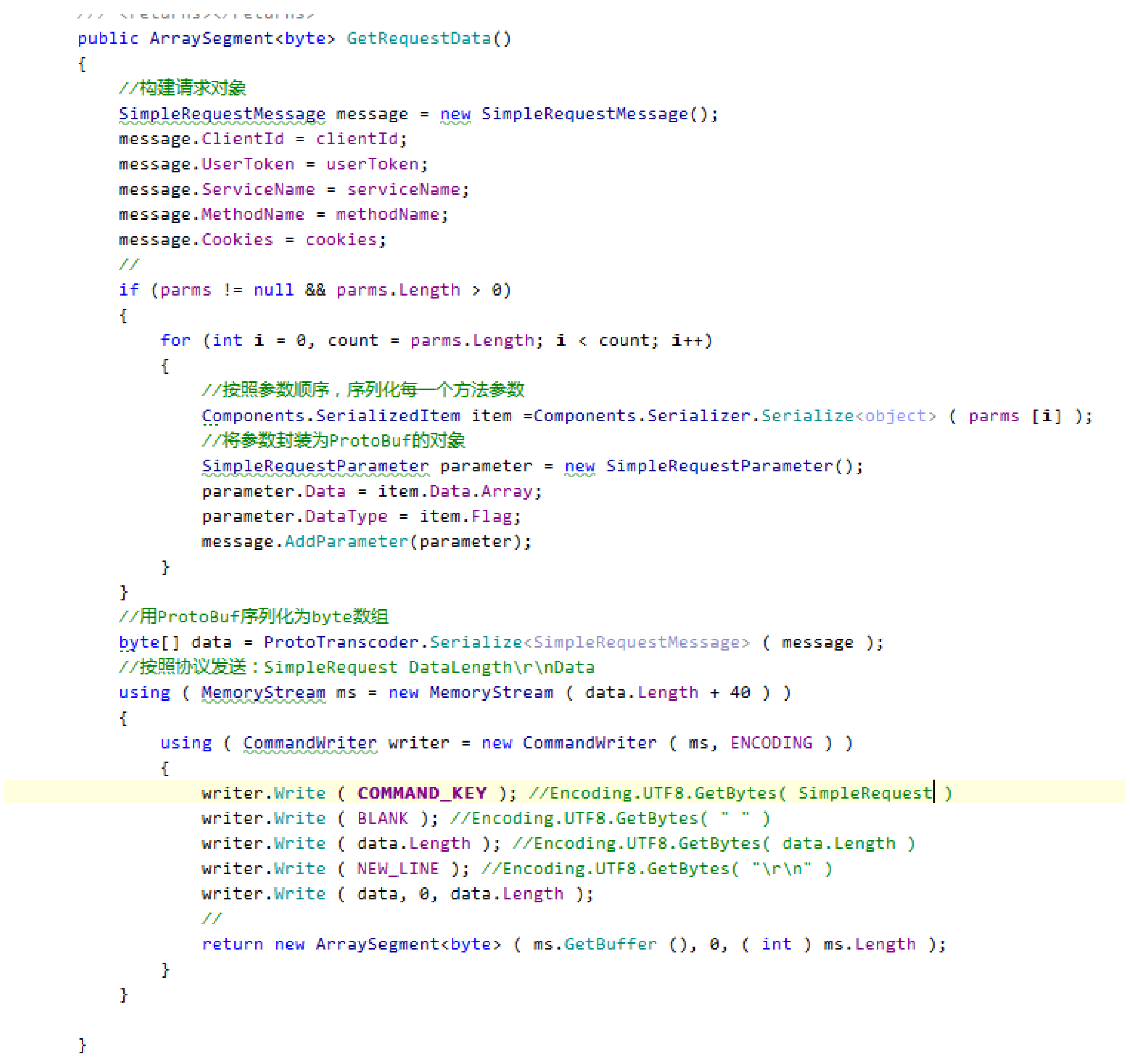
服务名称，ServiceName：UserService
方法名称，SignIn：SignIn
方法参数：4 个字符串参数



```
/// <summary>
/// 用户服务
/// </summary>
public class UserService : BaseService
{
    是否登录 IsSignIn

    #region 登录 SignIn
    /// <summary>
    /// 登录 SignIn
    /// </summary>
    /// <param name="emailOrMobile"></param>
    /// <param name="password"></param>
    /// <param name="vcodeId"></param>
    /// <param name="vcode"></param>
    /// <returns></returns>
    [AjaxMethod]
    public UserService_SignInResult SignIn(string emailOrMobile, string password, string vcodeId, string vcode)
    {
        if (string.IsNullOrEmpty(emailOrMobile))
        {
            throw new ArgumentException("请输入登录邮箱或手机号", "emailOrMobile");
        }
```

1.3.2 客户端（C#）生成发送内容

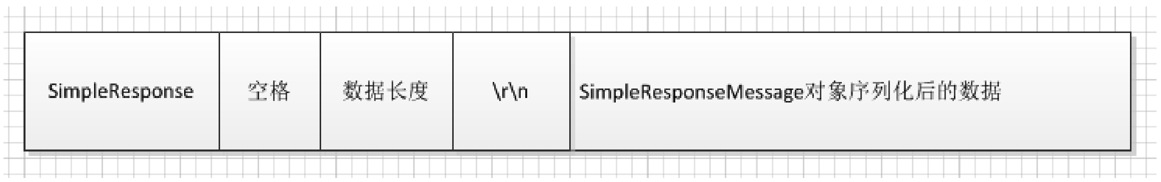


```
/// <returns></returns>
public ArraySegment<byte> GetRequestData()
{
    //构建请求对象
    SimpleRequestMessage message = new SimpleRequestMessage();
    message.ClientId = clientId;
    message.UserToken = userToken;
    message.ServiceName = serviceName;
    message.MethodName = methodName;
    message.Cookies = cookies;
    //
    if (parms != null && parms.Length > 0)
    {
        for (int i = 0, count = parms.Length; i < count; i++)
        {
            //按照参数顺序，序列化每一个方法参数
            Components.SerializedItem item =Components.Serializer.Serialize<object> ( parms [i] );
            //将参数封装为ProtoBuf的对象
            SimpleRequestParameter parameter = new SimpleRequestParameter();
            parameter.Data = item.Data.Array;
            parameter.DataType = item.Flag;
            message.AddParameter(parameter);
        }
    }
    //用ProtoBuf序列化为byte数组
    byte[] data = ProtoTranscoder.Serialize<SimpleRequestMessage> ( message );
    //按照协议发送：SimpleRequest DataLength\r\nData
    using ( MemoryStream ms = new MemoryStream ( data.Length + 40 ) )
    {
        using ( CommandWriter writer = new CommandWriter ( ms, ENCODING ) )
        {
            writer.Write ( COMMAND_KEY ); //Encoding.UTF8.GetBytes( SimpleRequest )
            writer.Write ( BLANK ); //Encoding.UTF8.GetBytes( " " )
            writer.Write ( data.Length ); //Encoding.UTF8.GetBytes( data.Length )
            writer.Write ( NEW_LINE ); //Encoding.UTF8.GetBytes( "\r\n" )
            writer.Write ( data, 0, data.Length );
            //
            return new ArraySegment<byte> ( ms.GetBuffer (), 0, ( int ) ms.Length );
        }
    }

}
```

2 接收

2.1 协议



2.2 SimpleResponseMessage

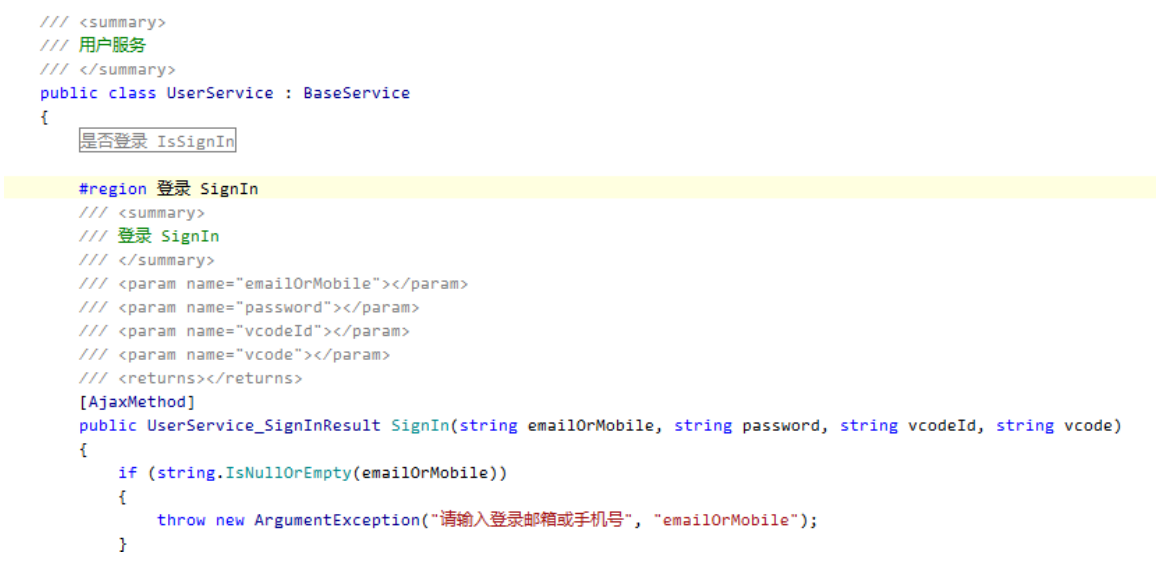
```
message SimpleResponseCookie {
        required string Name = 1;
        required string Value = 2;
        required int64 ExpireTime = 3;
}

message SimpleRequestResult {
        required int32 DataType = 1;
        required bytes Data = 2;
}

message SimpleResponseMessage {
        required bool Success = 1;
        required SimpleRequestResult Result = 2;
        optional string ErrorDesc = 3;
        required int64 ServerTime = 4;
        repeated SimpleResponseCookie Cookies = 5;
}
```

2.3 例子

2.3.1 服务器端



```
/// <summary>
/// 用户服务
/// </summary>
public class UserService : BaseService
{
    是否登录 IsSignIn

    #region 登录 SignIn
    /// <summary>
    /// 登录 SignIn
    /// </summary>
    /// <param name="emailOrMobile"></param>
    /// <param name="password"></param>
    /// <param name="vcodeId"></param>
    /// <param name="vcode"></param>
    /// <returns></returns>
    [AjaxMethod]
    public UserService_SignInResult SignIn(string emailOrMobile, string password, string vcodeId, string vcode)
    {
        if (string.IsNullOrEmpty(emailOrMobile))
        {
            throw new ArgumentException("请输入登录邮箱或手机号", "emailOrMobile");
        }
```

返回参数：UserService_SignInResult

定义：

```
message UserService_SignInResult {
        //用户ID
        optional int32 userId = 1;
        //昵称
        optional string nickname = 2;
        //头像
        optional string headPic = 3;
        //绑定手机号
        optional string mobile = 4;
        //微博Id
        optional int64 twitterId = 5;
        //状态
        optional int32 status = 6;
        //客服邮箱
        optional string serviceEmail = 7;
        //验证码ID
        optional string codeId = 8;
        //验证码图片地址
        optional string codeUrl = 9;
        //
        required bool success = 10;
        //
        optional string error = 11;
}
```

2.3.2 客户端（C#）获取返回内容

当接收到服务器端返回的全部数据（result）后：

```
SimpleResponseMessage responseMessage =
ProtoTranscoder.Deserialize<SimpleResponseMessage> ( result, 0, result.Length );
//获取真正的返回结果
SimpleRequestResult requestResult = responseMessage.Result;
Components.Serializer.Deserialize<T> ( new Components.SerializedItem
( requestResult.DataType, requestResult.Data ) )
```

*接收数据的过程：

```
                public AsyncSocketReceiveResult ProcessReceive(SocketAsyncEventArgs e)
                {
                        //保存本次Socket接收到的所有数据
                        stream2.Write ( e.Buffer, e.Offset, e.BytesTransferred );
                        //处理接收到的数据
                        StartAnalyse ();
                        //如果全部接收完成则退出，否则继续接收
                        if (isComplete)
                        {
                                return AsyncSocketReceiveResult.Complete;
                        }
                        return AsyncSocketReceiveResult.Continue;
                }

                private void StartAnalyse()
                {
                        if ( stream2.Length == 0 )
                                return;

                        switch ( _State )
                        {
                                //初始状态
                                case RemoteParseState.Action:
                                        ParseOnAction ();
                                        break;
                                //已接收到数据的总长度
                                case RemoteParseState.Length:
                                        ParseOnLength ();
                                        break;
                                //处理总长度和数据之间的回车换行符
                                case RemoteParseState.LF:
                                        ParseOnLF ();
                                        break;
                                //接收服务器端返回的数据
                                case RemoteParseState.Body:
                                        ParseOnBody ();
                                        break;
                        }
                }

                private void CheckMessageLength()
                {
                        if ( _MessageLength > DEFAULT_MAX_MESSAGE_LENGTH )
                                throw new Exception ( string.Format ( "消息超过最大允许长度: {0}",
DEFAULT_MAX_MESSAGE_LENGTH ) );
                }

                private void ParseOnAction()
                {
                        while ( stream2.Length > 0 )
                        {
```

```
                var b = ( byte ) stream2.ReadByte ();
                if ( b == ' ' )
                {
                    _Action = Encoding.UTF8.GetString ( _Buffer, 0, _Index );
                    if ( string.Compare ( _Action, REMOTESERVICE_REQUEST_TAG,
StringComparison.OrdinalIgnoreCase ) != 0 )
                    {
                        throw new Exception ( string.Format ( "不支持的Command: {0}",
_Action ) );
                    }

                    _State = RemoteParseState.Length;
                    _Index = 0;
                    break;
                }
                else
                {
                    _Buffer [_Index] = b;
                    _Index++;
                }
            }

            if ( stream2.Length > 0 )
                ParseOnLength ();
        }

        private void ParseOnLength()
        {
            while ( stream2.Length > 0 )
            {
                var b = ( byte ) stream2.ReadByte ();
                if ( b == '\r' )
                {
                    var length = Encoding.UTF8.GetString ( _Buffer, 0, _Index );
                    _MessageLength = SafeConvert.ToInt32 ( length );
                    CheckMessageLength ();
                    _State = RemoteParseState.LF;
                    _Index = 0;
                    break;
                }
                else
                {
                    _Buffer [_Index] = b;
                    _Index++;
                }
            }

            if ( stream2.Length > 0 )
                ParseOnLF ();
        }

        private void ParseOnLF()
        {
            stream2.ReadByte ();
            _State = RemoteParseState.Body;

            if ( stream2.Length > 0 )
                ParseOnBody ();
        }

        private void ParseOnBody()
        {
            if ( stream2.Length >= _MessageLength )
            {
                byte [] bits = new byte [_MessageLength];
                stream2.Read ( bits, 0, bits.Length );
                //
                _Message = bits;
                //
                _LastActivityTime = DateTime.Now;
                //
                _IsMessageReady = true;
                //
                isComplete = true;
            }
        }
```

3 序列化

默认采用 Little-endian，即最大的有效字节位于单词的右端

3.1 支持的类型

3.1.1 8 种基本类型

类型名称	DataType 枚举值	序列化参考	反序列化参考
ByteArray	1		
bool	10	=> new byte [1] { value ? 1 : 0 }	bytes [0] == 1
int32	4	new byte[4]	(int) *numPtr | (int) numPtr[1] << 8 | (int) numPtr[2] << 16 | (int) numPtr[3] << 24
int64	5	new byte[8]	(long) (uint) ((int) *numPtr | (int) numPtr[1] << 8 | (int) numPtr[2] << 16 | (int) numPtr[3] << 24) | (long) ((int) numPtr[4] | (int) numPtr[5] << 8 | (int) numPtr[6] << 16 | (int) numPtr[7] << 24) << 32
Float	18	new byte[4]	同 int32
Double	19	new byte[8]	同 int64
String	3	UTF8 编码	
ProtoBuf	21		
*NULL 对象	0	new byte [] { 0 }	

3.1.2 Gzip 压缩的类型（byte 数组>100K）

类型名称	枚举值	序列化参考	反序列化参考
CompressedByteArray	255		
CompressedString	254		
CompressedProtoBuf	251		

3.2 类型传递

3.2.1 序列化例子（C#）

根据对象类型获取 DataType 枚举
将对象序列化为 byte 数组

```
const uint COMPRESSION_THRESHOLD = 100 * 1024; //100K

public static SerializedItem Serialize<T>( T value )
{
    return Serialize<T> ( value, COMPRESSION_THRESHOLD );
}

public static SerializedItem Serialize<T>( object value, uint compressionThreshold )
{
    SerializedType type = SerializedType.Null;
    byte [] bytes;
    if ( value == null )
    {
        type = SerializedType.Null;
        bytes = new byte [] { 0 };
    }
    else if ( value is byte [] )
    {
        bytes = ( byte [] ) value;
        type = SerializedType.ByteArray;
        if ( bytes.Length > compressionThreshold )
        {
            bytes = SerializeHelper.Compress ( bytes );
            type = SerializedType.CompressedByteArray;
        }
    }
    else if ( value is string )
    {
        bytes = Encoding.UTF8.GetBytes ( ( string ) value );
        type = SerializedType.String;
        if ( bytes.Length > compressionThreshold )
        {
            bytes = SerializeHelper.Compress ( bytes );
            type = SerializedType.CompressedString;
        }
    }
    else if ( value is bool )
    {
        bytes = new byte [] { ( byte ) ( ( bool ) value ? 1 : 0 ) };
        type = SerializedType.Bool;
    }
    else if ( value is int )
    {
        bytes = BitConverter.GetBytes ( ( int ) value );
        type = SerializedType.Int32;
    }
    else if ( value is long )
    {
        bytes = BitConverter.GetBytes ( ( long ) value );
        type = SerializedType.Int64;
    }
    else if ( value is float )
    {
        bytes = BitConverter.GetBytes ( ( float ) value );
        type = SerializedType.Float;
    }
    else if ( value is double )
    {
        bytes = BitConverter.GetBytes ( ( double ) value );
        type = SerializedType.Double;
    }
    else
    {
        bytes = ProtoTranscoder.Serialize<T> ( ( T ) value );
        type = SerializedType.ProtoBuf;
        if ( bytes.Length > compressionThreshold )
        {
            bytes = SerializeHelper.Compress ( bytes );
            type = SerializedType.CompressedProtoBuf;
        }
    }
    //
    ArraySegment<byte> data = new ArraySegment<byte> ( bytes, 0, bytes.Length );
    return new SerializedItem ( ( int ) type, data );
}
```

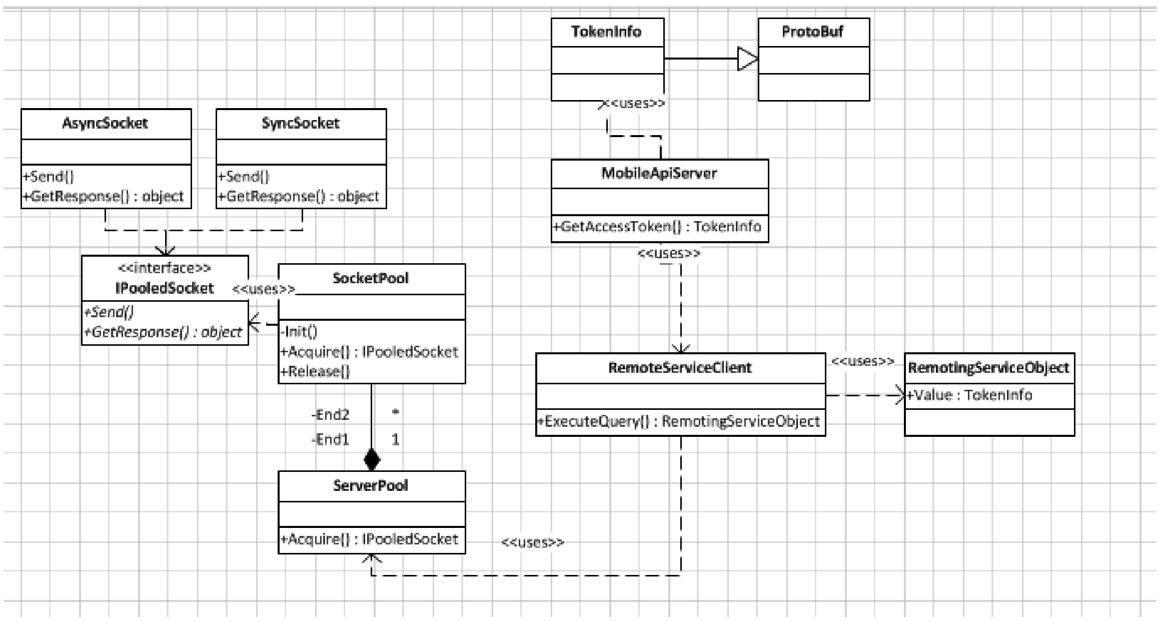
3.2.2 反序列化例子（C#）

通过 DataType 枚举值来标识数据的的类型
根据数据的类型反序列化为对象

```
public static T Deserialize<T>( SerializedItem item )
{
    SerializedType type = ( SerializedType ) item.Flag;
    byte [] bytes = item.Data.Array;
    int offset = item.Data.Offset;
    int count = item.Data.Count;
    //
    return ( T ) Deserialize<T> ( type, bytes, offset, count );
}

static object Deserialize<T>( SerializedType serializedType, byte [] bytes, int offset, int
count )
{
    byte [] bits;
    switch ( serializedType )
    {
        case SerializedType.Null:
            return null;
        case SerializedType.String:
            return Encoding.UTF8.GetString ( bytes, offset, count );
        case SerializedType.Bool:
            return bytes [0] == 1;
        case SerializedType.Int32:
            return BitConverter.ToInt32 ( bytes, offset );
        case SerializedType.Int64:
            return BitConverter.ToInt64 ( bytes, offset );
        case SerializedType.Float:
            return BitConverter.ToSingle ( bytes, offset );
        case SerializedType.Double:
            return BitConverter.ToDouble ( bytes, offset );
        case SerializedType.ProtoBuf:
            using ( MemoryStream ms = new MemoryStream ( bytes, offset, count ) )
            {
                return ProtoBuf.Serializer.Deserialize<T> ( ms );
            }
        case SerializedType.CompressedByteArray:
            bits = SerializeHelper.Decompress ( bytes );
            return Deserialize<T> ( SerializedType.ByteArray, bits, 0, bits.Length );
        case SerializedType.CompressedString:
            bits = SerializeHelper.Decompress ( bytes );
            return Deserialize<T> ( SerializedType.String, bits, 0, bits.Length );
        case SerializedType.CompressedProtoBuf:
            bits = SerializeHelper.Decompress ( bytes );
            return Deserialize<T> ( SerializedType.ProtoBuf, bits, 0, bits.Length );
        case SerializedType.ByteArray:
        default:
            if ( bytes.Length == count )
                return bytes;
            byte [] retval = new byte [count];
            Buffer.BlockCopy ( bytes, offset, retval, 0, count );
            return retval;
    }
}
```

4 客户端架构



1、接口IPooledSocket定义了同步和异步socket的主要方法：Send和GetResponse

2、SocketPool初始化时根据需要实例化一个队列，队列里的是具体的IPooledSocket（如：SyncSocket）

3、SocketPool对外暴露两个主要方法：Acquire和Release，分别是获取IPooledSocket的实例和往连接池还回已使用完成的实例。

4、ServerPool用于处理同一个Server保持多个SocketPool的情况（可以省略）

5、RemoteServiceClient 用于接受客户端实际的方法调用，服务器端返回一个具体的 RemotingServiceObject，里面的 Value 用于保存具体的实体对象（ProtoBuf 对象）