# The Built-in Common Format Protocol Implementation Templates

English (United States) ▾ v1.6 ▾

**Keywords**: Protocol Tools, Custom Protocol, TerminatorReceiveFilter, CountSpliterReceiveFilter, FixedSizeReceiveFilter, BeginEndMarkReceiveFilter, FixedHeaderReceiveFilter

After reading the previous document, you probably find implementing your own protocol using SuperSocket probably is not easy for you. To make this job easier, SuperSocket provides some common protocol tools, which you can use to build your own protocol easily and fast:

- **TerminatorReceiveFilter** (SuperSocket.SocketBase.Protocol.TerminatorReceiveFilter, SuperSocket.SocketBase)
- **CountSpliterReceiveFilter** (SuperSocket.Facility.Protocol.CountSpliterReceiveFilter, SuperSocket.Facility)
- **FixedSizeReceiveFilter** (SuperSocket.Facility.Protocol.FixedSizeReceiveFilter, SuperSocket.Facility)
- **BeginEndMarkReceiveFilter** (SuperSocket.Facility.Protocol.BeginEndMarkReceiveFilter, SuperSocket.Facility)
- **FixedHeaderReceiveFilter** (SuperSocket.Facility.Protocol.FixedHeaderReceiveFilter, SuperSocket.Facility)

**facility : 설비, 시설, 편의, 쉬움, 기지**

## TerminatorReceiveFilter - Terminator Protocol

Similar with command line protocol, some protocols use a terminator to identify a request. For example, one protocol uses two chars "##" as terminator, then you can use the class "TerminatorReceiveFilterFactory":

```
/// <summary>
/// TerminatorProtocolServer
/// Each request end with the terminator "##"
/// ECHO Your message##
/// </summary>
public class TerminatorProtocolServer : AppServer
{
    public TerminatorProtocolServer()
        : base(new TerminatorReceiveFilterFactory("##"))
    {

    }
}
```

The default RequestInfo is StringRequestInfo, you also can create your own RequestInfo class, but it requires a bit more work:

Implement your ReceiveFilter base on TerminatorReceiveFilter:

```
public class YourReceiveFilter : TerminatorReceiveFilter<YourRequestInfo>
{
    //More code
}
```

Implement your ReceiveFilterFactory which can create your request filter instances:

```
public class YourReceiveFilterFactory : IReceiveFilterFactory<YourRequestInfo>
{
    //More code
}
```

And then use the request filter factory in your AppServer.

## CountSpliterReceiveFilter - Fixed Number Split Parts with Separator Protocol

Some protocols defines their requests look like in the format of "#part1#part2#part3#part4#part5#part6#part7#". There are 7 parts in one request and all parts are separated by char '#'. This kind protocol's implementing also is quite easy:

```
/// <summary>
/// Your protocol likes like the format below:
/// #part1#part2#part3#part4#part5#part6#part7#
/// </summary>
public class CountSpliterAppServer : AppServer
{
    public CountSpliterAppServer()
        : base(new CountSpliterReceiveFilterFactory((byte)'#', 8)) // 7 parts but 8 separators
    {

    }
}
```

You also can customize your protocol deeper using the classes below:

```
CountSpliterReceiveFilter<TRequestInfo>
CountSpliterReceiveFilterFactory<TReceiveFilter>
CountSpliterReceiveFilterFactory<TReceiveFilter, TRequestInfo>
```

## FixedSizeReceiveFilter - Fixed Size Request Protocol

In this kind protocol, the size of all requests are same. If your each request is 9 characters string like "KILL BILL", what you should do is implementing a ReceiveFilter like the code below:

```
class MyReceiveFilter : FixedSizeReceiveFilter<StringRequestInfo>
{
    public MyReceiveFilter()
        : base(9) //pass in the fixed request size
    {

    }

    protected override StringRequestInfo ProcessMatchedRequest(byte[] buffer, int offset, int length, bool toBeCopied)
    {
        //TODO: construct the request info instance from the parsed data and then return
    }
}
```

Then use the receive filter in your AppServer class:

```
public class MyAppServer : AppServer
{
    public MyAppServer()
        : base(new DefaultReceiveFilterFactory<MyReceiveFilter, StringRequestInfo>()) //using default receive filter factory
    {

    }
}
```

## BeginEndMarkReceiveFilter - The Protocol with Begin and End Mark

Every message in this protocol have fixed begin mark and end mark. For example, I have a protocol all messages are in the format "!xxxxxxxxxxxxxx$". In this case "!" is begin mark and the "$" is end mark, so my receive filter looks like:

```
class MyReceiveFilter : BeginEndMarkReceiveFilter<StringRequestInfo>
{
    //Both begin mark and end mark can be two or more bytes
    private readonly static byte[] BeginMark = new byte[] { (byte)'!' };
    private readonly static byte[] EndMark = new byte[] { (byte)'$' };

    public MyReceiveFilter()
        : base(BeginMark, EndMark) //pass in the begin mark and end mark
    {

    }

    protected override StringRequestInfo ProcessMatchedRequest(byte[] readBuffer, int offset, int length)
    {
        //TODO: construct the request info instance from the parsed data and then return
    }
}
```

Then use the receive filter in your AppServer class:

```
public class MyAppServer : AppServer
{
    public MyAppServer()
        : base(new DefaultReceiveFilterFactory<MyReceiveFilter, StringRequestInfo>()) //using default receive filter factory
    {

    }
}
```

## FixedHeaderReceiveFilter - Fixed Header with Body Length Protocol

This kind protocol defines each request has two parts, the first part contains some basic information of this request include the length of the second part. We usually call the first part is header and the second part is body.

For example, we have a protocol like that: the header contains 6 bytes, the first 4 bytes represent the request's name, the last 2 bytes represent the length of the body:

```
/// +-------+---+-------------------------------+
/// |request| l |                               |
/// | name  | e |    request body               |
/// |  (4)  | n |                               |
/// |       |(2)|                               |
/// +-------+---+-------------------------------+
```

Using SuperSocket, you can implement this kind protocol easily:

```
class MyReceiveFilter : FixedHeaderReceiveFilter<BinaryRequestInfo>
{
    public MyReceiveFilter()
        : base(6)
    {

    }

    protected override int GetBodyLengthFromHeader(byte[] header, int offset, int length)
    {
        return (int)header[offset + 4] * 256 + (int)header[offset + 5];
    }

    protected override BinaryRequestInfo ResolveRequestInfo(ArraySegment<byte> header, byte[] bodyBuffer, int offset, int length)
    {
        return new BinaryRequestInfo(Encoding.UTF8.GetString(header.Array, header.Offset, 4), bodyBuffer.CloneRange(offset, length));
    }
}
```

You need to implement your own request filter base on FixedHeaderReceiveFilter.

- The number 6 passed into the parent class's constructor means the size of the request header;
- The method "GetBodyLengthFromHeader(...)" you should override returns the length of the body according the received header;
- the method "ResolveRequestInfo(....)" you should override returns the RequestInfo instance according the received header and body.

Then you can build a receive filter factory or use the default receive factory to use this receive filter in SuperSocket.

- Prev: The Built-in Command Line Protocol (/v1-6/en-US/The-Built-in-Command-Line-Protocol)
- Next: Implement Your Own Communication Protocol with IRequestInfo, IReceiveFilter and etc (/v1-6/en-US/Implement-Your-Own-Communication-Protocol-with-IRequestInfo,-IReceiveFilter-and-etc)