

[翻译]erlang分发协议-erl distribution protocol (OTP 25)



原文链接: https://www.erlang.org/doc/apps/erts/erl_dist_protocol.html

源码doc路径: erts/doc/doc/src/erl_dist_protocol.xml

13 Distrubution协议

该描述尚未完善. 如果协议更新, 该描述将被更新。然而多年来, 无论是erlang port mapper daemon (epmd) 还是erlang节点通信协议, 都是稳定的。

分发协议可以分为四个部分:

- 低级socket连接 (1)
- 握手、交换节点名称和身份验证 (2)
- 身份验证 (由net_kernel(3)完成) (3)
- 已连接 (4)

一个节点通过EPMD (在另一个主机) 获取另一个节点的端口号来发起连接请求。

对于每个运行分布式 Erlang 节点的主机, EPMD 也将运行。作为 Erlang 节点启动的结果, EPMD 可以显式启动或自动启动。

默认情况下, EPMD 侦听端口 4369。

上面的 (3) 和 (4) 同时进行, 但是如果net_kernel 使用无效的 cookie 进行通信 (1 秒后), 它会断开另一个节点的连接。

所有多字节字段中的整数都按大端存储。

! Erlang 分发协议本身并不安全, 也不打算这样做。为了获得安全分发, 分布式节点应配置为使用 tls 上的分发。有关如何设置安全分布式节点的详细信息, 请参阅 [Using SSL for Erlang Distribution User's Guide](#)。

13.1 EPMD 协议

下图展示了 EPMD请求过程。.

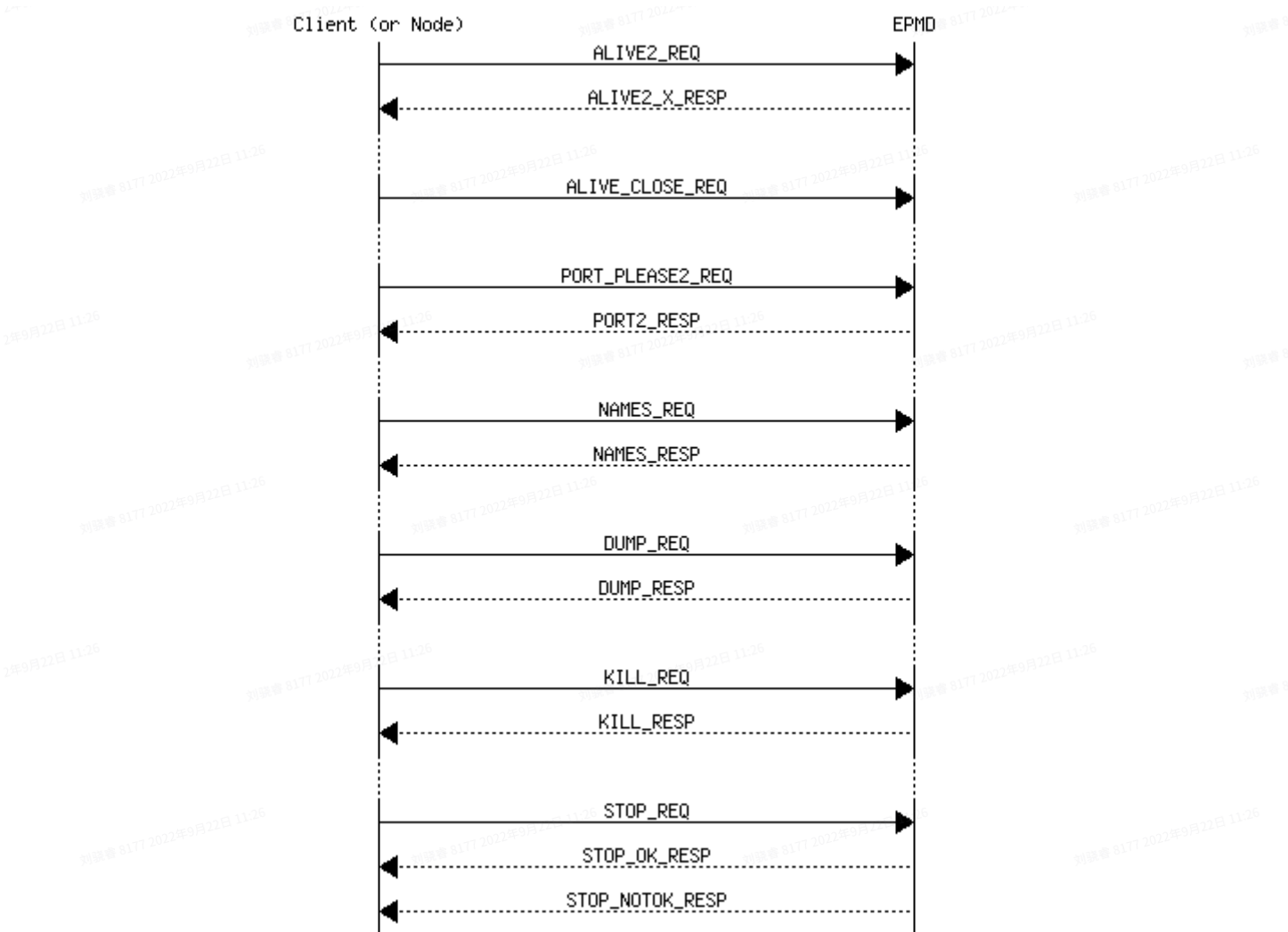


Figure 13.1: Summary of EPMD Requests

每个请求*_REQ前面都有一个 2 字节的长度字段。因此，整体请求格式如下：

	A	B
1	2	n
2	Length	Request

Table 13.1: Request Format

在 EPMD 中注册一个节点

当一个分布式节点启动时，它会在 EPMD 中注册自己。下面描述的消息ALIVE2_REQ从节点发送到 EPMD。来自 EPMD 的响应是ALIVE2_X_RESP（或 ALIVE2_RESP）。

	A	B	C	D	E	F	G
1	1	2	1	1	2	2	2
2	120	PortNo	NodeType	Protocol	HighestVersion	LowestVersion	Nlen

Table 13.2: ALIVE2_REQ (120)

PortNo

节点接受连接请求的端口号。

NodeType

77 = 普通 Erlang 节点，72 = 隐藏节点 (C-node) , ...

Protocol

0 = TCP/IPv4, ...

HighestVersion

此节点可以处理的最高分发协议版本。OTP 23 及更高版本中的值为 6。旧节点仅支持版本 5。

LowestVersion

此节点可以处理的最低分发版本。应该是 5 以支持连接到早于 OTP 23 的节点。

Nlen

字段NodeName的长度（以字节为单位）。

NodeName

节点名称为Nlen个字节的 UTF-8 编码字符串。

Elen

字段Extra的长度。

Extra

Elen字节的Extra字段。

只要节点是分布式节点，就必须保留创建到 EPMD 的连接。当连接关闭时，节点会自动从 EPMD 注销。

响应消息是ALIVE2_X_RESP或 ALIVE2_RESP，具体取决于分发版本。如果节点和 EPMD 都支持分发版本 6，则响应为 ALIVE2_X_RESP否则为较旧的ALIVE2_RESP：

	A	B	C
1	1	1	4
2	118	Result	Creation

Table 13.3: ALIVE2_X_RESP (118) with 32 bit creation

	A	B	C
1	1	1	2
2	121	Result	Creation

Table 13.4: ALIVE2_RESP (121) with 16-bit creation

结果 = 0 -> 正常，结果 > 0 -> 错误。

从 EPMD 注销节点

节点通过关闭与在节点注册时建立的 EPMD 的 TCP 连接来从 EPMD 注销自己。

获取另一个节点的distribution port

当一个节点想要连接到另一个节点时，它首先向节点所在主机上的 EPMD发出 PORT_PLEASE2_REQ请求，以获取节点侦听的分发端口。

	A	B
1	1	N
2	122	NodeName

Table 13.5: PORT_PLEASE2_REQ (122)

其中 N =长度- 1。

	A	B
1	1	1
2	119	Result

Table 13.6: PORT2_RESP (119) Response Indicating Error, Result > 0

or

	A	B	C	D	E	F	G
1	1	1	2	1	1	2	2
2	119	Result	PortNo	NodeType	Protocol	HighestVersion	LowestVer

Table 13.7: PORT2_RESP, Result = 0

如果Result > 0，则数据包仅包含 [119, Result]。

EPMD 在发送信息后关闭套接字。

从 EPMD 获取所有注册名称

此请求通过 Erlang 函数 `net_adm:names/1,2`使用。与 EPMD 建立 TCP 连接并发送此请求。

	A
1	1
2	110

Table 13.8: NAMES_REQ (110)

NAMES_REQ的响应如下

	A	B
1	4	
2	EPMDPortNo	NodeInfo*

Table 13.9: NAMES_RESP

NodeInfo是为每个活动节点编写的字符串。写入所有NodeInfo后，EPMD 将关闭连接。

NodeInfo的格式用 Erlang 表示如下：

```
1 io:format( "name ~ts at port ~p~n" , [NodeName, Port]).
```

转储 EPMD 中的所有数据

这个请求并没有真正使用，它被视为一个调试功能。

	A
1	1
2	100

Table 13.10: DUMP_REQ

DUMP_REQ的响应如下：

	A	B
1	4	
2	EPMDPortNo	NodeInfo*

Table 13.11: DUMP_RESP

NodeInfo是为保存在 EPMD 中的每个节点编写的字符串。写入所有NodeInfo后，EPMD 将关闭连接。

NodeInfo是，用 Erlang 表示：

```
1 io:format("active name ~ts at port ~p, fd = ~p~n",[NodeName, Port, Fd]).
```

或

```
1 io:format("old/unused name ~ts at port ~p, fd = ~p ~n",[NodeName, Port, Fd]).
```

Kill EPMD

此请求会终止正在运行的 EPMD。它几乎从未使用过。

	A
1	1
2	107

Table 13.12: KILL_REQ

KILL_REQ的响应如下：

	A
1	2
2	OKString

Table 13.13: KILL_RESP

其中OKString是"OK".

STOP_REQ（未启用）

	A	B
1	1	n
2	115	NodeName

Table 13.14: STOP_REQ

其中n = Length - 1.

STOP_REQ的响应如下

	A
1	7
2	OKString

Table 13.15: STOP_RESP

其中OKString是"STOPPED".

失败响应如下所示：

	A
1	7
2	NOKString

Table 13.16: STOP_NOTOK_RESP

其中NOKString是"NOEXIST".

13.2 Distribution 握手

本节介绍节点之间用于建立连接的分发握手协议。该协议是在 Erlang/OTP R6 中引入的，并在 OTP 23 中进行了修改。从 OTP 25 开始，对旧协议的支持被删除了。因此，OTP 25 节点无法连接到早于 OTP 23 的节点。本文档仅描述了新修订的协议。

概述

TCP/IP 分发使用需要基于连接的协议的握手，也就是说，**该协议在握手过程之后不包括任何身份验证。**

这并不完全安全，因为它容易受到接管攻击，但它是公平安全和性能之间的权衡。

cookie 从不以明文形式发送，并且握手过程期望客户端（称为A）是第一个证明它可以生成足够摘要的客户端。摘要是使用 MD5 消息摘要算法生成的，challenges 是随机数。

定义

challenge 是大端顺序的 32 位整数。下面的函数 `gen_challenge()` 返回一个随机的 32 位整数，用作挑战。

digest是challenge (text) 与 cookie (text) 共同生成的MD5 哈希 (16byte) 。digest由函数 `gen_digest(Challenge, Cookie)` 生成。

out_cookie是用于与某个节点通信的出向cookie，A的out_cookie 与B的in_cookie对应，反之亦然。A的out_cookie和A的 in_cookie**不必相同**。下图的out_cookie(Node)代表Node的out_cookie。

in_cookie是另一个节点在与我们通信时预期使用的 cookie，因此A的in_cookie对应于 B的 out_cookie。下图的in_cookie(Node)代表Node的in_cookie。

cookie 是可以被视为密码的文本字符串。

握手中的每条消息都以一个 16 位大端整数开头，代表消息长度（不包括两个字节的长度位）。在 Erlang 中，这对应于 `gen_tcp(3)` 中的选项{packet, 2}。请注意，在握手之后，distribution切换到 4 字节数据包头。（笔者注：即握手包使用16bit表示包体长度，通信包使用32bit表示包体长度）。

握手细节

想象两个节点，A发起握手，B 接受连接。

1) connect/accept

A通过 TCP/IP连接到B，B接受连接。

2) send_name/receive_name

A向B发送初始标识。该消息如下所示（代表长度的数据包标头已删除）：

	A	B	C	D	E
1	1	8	4	2	Nlen
2	'N'	Flags	Creation	Nlen	Name

Table 13.17: New send_name ('N') for protocol version 6

Flags

节点 A 的 64 位大端的Flag位

Creation

节点A用来创建其 pid、ports和references的节点化身标识符。

Name

A的完整节点名称，作为字节字符串。

Nlen

16 位大端的node name的字节长度。

必须接受和忽略 node name之后的任何额外数据。

如果 DFLAG_NAME_ME 标志位被set，Name必须是不含@的hostname。

3) recv_status/send_status

B向A发送状态消息，指示是否允许连接。

	A	B
1	1	Slen
2	's'	Status

Table 13.18: The format of the status message

's' 是命令标志。Status是作为字符串的状态代码（不是以 null 结尾的）。定义了以下状态码：

ok

握手将继续。

ok_simultaneous

握手将继续，但A被告知 B有另一个正在进行的连接尝试将被关闭（同时连接，其中A的名称大于B的名称，比较字面意思）。

nok

握手不会继续，因为B已经有一个正在进行的握手，它自己已经启动了（同时连接B的名称大于A的名称）。

not_allowed

由于某些（未指定）安全原因，不允许连接。.

alive

握手将继续，但A通过设置标志DFLAG_NAME_ME请求动态节点名称。A的动态节点名称 在来自B的状态消息的末尾提供。 在send_name中作为Name发送的A的主机名将被节点B用来生成完整的动态节点名。

named:

握手将继续，但A通过设置标志DFLAG_NAME_ME请求动态节点名称。A的动态节点名称 在来自B的状态消息的末尾提供。 在send_name中作为Name发送的A的主机名将被节点B用来生成完整的动态节点名。

	A	B	C	D	E
1	1	Slen=6	2	Nlen	4
2	's'	Status='named:'	Nlen	Name	Creation

Table 13.19: The format of the 'named:' status message

Name

A的完整动态节点名称，作为字节字符串

Nlen

16 位大端的节点名称的字节长度。

Creation

节点B生成的节点A的化身标识符。
必须接受和忽略节点Creation后的任何额外数据。

3B) send_status/recv_status

如果 status 是alive，节点A会回复另一个状态消息，其中包含true，表示连接将继续（来自该节点的旧连接已断开），或false，表示连接将关闭（连接尝试错误）。

4) recv_challenge/send_challenge

如果状态为ok或ok_simultaneous，则握手继续，B向A发送另一条消息，即challenge。challenge包含与最初从A发送到B的“name”消息相同类型的信息，外加一个 32 位challenge：

	A	B	C	D	E	F
1	1	8	4	4	2	Nlen
2	'N'	Flags	Challenge	Creation	Nlen	Name

Table 13.20: The new challenge message format (version 6)

Challenge是一个 32 位大端整数。其他字段是节点B的flags、creation 和完整node name，类似于send_name消息。必须接受和忽略node name之后的任何额外数据。

5) send_challenge_reply/recv_challenge_reply

现在A生成了一个digest和它自己的challenge。这些被一起打包发送到B：

	A	B	C
1	1	4	16
2	'r'	Challenge	Digest

Table 13.21: The challenge_reply message

challenge是A对B的challenge。digest是A从上一步B发送的challenge构造的 MD5 摘要。

6) recv_challenge_ack/send_challenge_ack

B检查从A收到的digest是否正确，并根据从A收到的challenge生成digest。然后将digest发送到A。消息如下：

	A	B
1	1	16
2	'a'	Digest

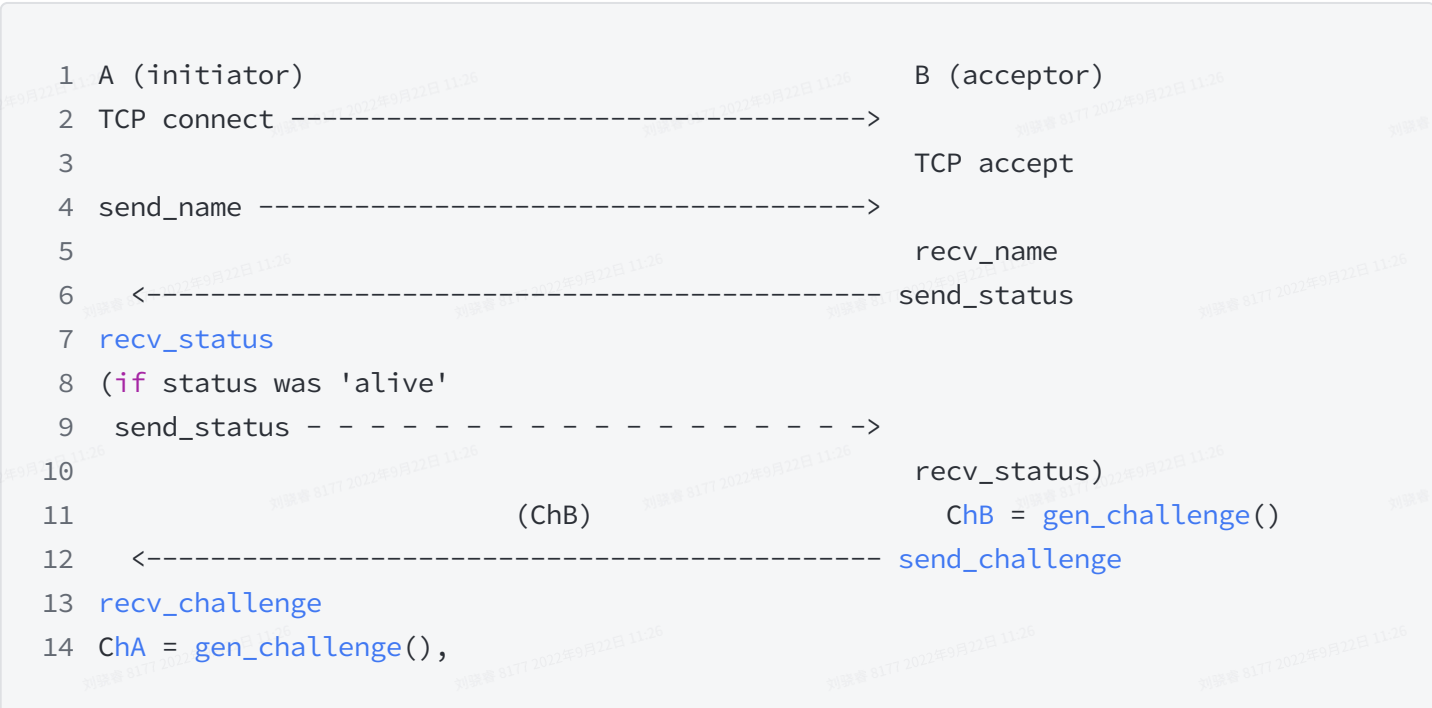
Table 13.22: The challenge_ack message

Digest is the digest calculated by B for A's challenge.

7) check

A检查来自B的digest并且连接已建立。

Semigraphic View



```

15 OCA = out_cookie(B),
16 DiA = gen_digest(ChB, OCA)
17                                     (ChA, DiA)
18 send_challenge_reply ----->
19                                     recv_challenge_reply
20                                     ICB = in_cookie(A),
21                                     check:
22                                     DiA == gen_digest (ChB, ICB)?
23                                     - if OK:
24                                     OCB = out_cookie(A),
25                                     DiB = gen_digest (ChA, OCB)
26                                     (DiB)
27 <----- send_challenge_ack
28 recv_challenge_ack               DONE
29 ICA = in_cookie(B),             - else:
30 check:                           CLOSE
31 DiB == gen_digest(ChA, ICA)?
32 - if OK:
33     DONE
34 - else:
35     CLOSE

```

Distribution Flags

在分发握手的早期，两个参与节点交换能力标志。这样做是为了确定应该如何执行两个节点之间的通信。两个节点提供的能力的交集定义了将要使用的能力。功能标志定义如下：

-define(DFLAG_PUBLISHED,16#1).

该节点将被发布并且是全局命名空间的一部分。

-define(DFLAG_ATOM_CACHE,16#2).

该节点实现了一个原子缓存（已过时）。

-define(DFLAG_EXTENDED_REFERENCES,16#4).

该节点实现扩展（ 3×32 位）引用。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_DIST_MONITOR,16#8).

节点实现分布式进程监控。

-define(DFLAG_FUN_TAGS,16#10).

该节点在分发协议中为 funs (lambdas) 使用单独的标签。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_DIST_MONITOR_NAME,16#20).

节点实现分布式命名进程监控。

-define(DFLAG_HIDDEN_ATOM_CACHE,16#40).

(隐藏) 节点实现原子缓存 (已过时)。

-define(DFLAG_NEW_FUN_TAGS,16#80).

节点理解NEW_FUN_EXT标签。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_EXTENDED_PIDS_PORTS,16#100).

该节点可以处理扩展的 pid 和端口。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_EXPORT_PTR_TAG,16#200).

该节点理解EXPORT_EXT标记。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_BIT_BINARIES,16#400).

节点理解BIT_BINARY_EXT标签。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_NEW_FLOATS,16#800).

节点理解NEW_FLOAT_EXT标签。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_UNICODE_IO,16#1000).

-define(DFLAG_DIST_HDR_ATOM_CACHE,16#2000).

节点在distribution header中实现原子缓存。

-define(DFLAG_SMALL_ATOM_TAGS, 16#4000).

节点理解SMALL_ATOM_EXT标签。

-define(DFLAG_UTF8_ATOMS, 16#10000).

该节点理解用 ATOM_UTF8_EXT和 SMALL_ATOM_UTF8_EXT编码的 UTF-8 原子。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_MAP_TAG, 16#20000).

该节点理解地图标签 MAP_EXT。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_BIG_CREATION, 16#40000).

该节点理解大节点创建标签 NEW_PID_EXT、NEW_PORT_EXT和 NEWER_REFERENCE_EXT。该标志是强制性的。如果不存在，则拒绝连接。

-define(DFLAG_SEND_SENDER, 16#80000).

使用SEND_SENDER control message 代替SEND控制消息，并使用 SEND_SENDER_TT控制消息代替SEND_TT控制消息。

-define(DFLAG_BIG_SEQTRACE_LABELS, 16#100000).

该节点将任何术语理解为 seqtrace 标签。

`-define(DFLAG_EXIT_PAYLOAD, 16#400000).`

使用PAYLOAD_EXIT、PAYLOAD_EXIT_TT、PAYLOAD_EXIT2、PAYLOAD_EXIT2_TT 和 PAYLOAD_MONITOR_P_EXIT **control message** 而不是非 PAYLOAD 变体。

`-define(DFLAG_FRAGMENTS, 16#800000).`

使用 **fragmented** distribution messages 发送大消息

`-define(DFLAG_HANDSHAKE_23, 16#1000000).`

该节点支持 OTP 23 中引入的新连接建立握手（版本 6）。此标志是强制性的（来自 OTP 25）。如果不存在，则拒绝连接。

`-define(DFLAG_UNLINK_ID, 16#2000000).`

使用 **new link protocol**.



此标志将在 OTP 26 中成为强制性标志。

除非两个节点都设置了 DFLAG_UNLINK_ID 标志，否则 **old link protocol** 将用作后备。

`-define(DFLAG_MANDATORY_25_DIGEST, (1 bsl 36)).`

该节点支持 OTP 25 中强制的所有功能。在 OTP 25 中引入。



此标志将在 OTP 27 中成为强制性标志。

`-define(DFLAG_SPAWN, (1 bsl 32)).`

设置是否支持 **SPAWN_REQUEST**、**SPAWN_REQUEST_TT**、**SPAWN_REPLY**、**SPAWN_REPLY_TT** 控制消息。

`-define(DFLAG_NAME_ME, (1 bsl 33)).`

动态节点名称。这不是一种能力，而是用作来自连接节点的请求，以从接受节点接收其节点名称作为握手的一部分。

`-define(DFLAG_V4_NC, (1 bsl 34)).`

节点接受 pid、端口和引用（节点容器类型版本 4）中的大量数据。在 pid 情况下，NEW_PID_EXT 中的完整 32 位 ID 和 Serial 字段，在端口情况下，**V4_PORT_EXT** 中的 64 位整数，在参考情况下，**NEWER_REFERENCE_EXT** 现在最多接受 5 个 32 位 ID 字。在 OTP 24 中引入。



此标志将在 OTP 26 中成为强制性标志。

`-define(DFLAG_ALIAS, (1 bsl 35)).`

节点支持进程别名，并且可以通过它处理 `ALIAS_SEND` 和 `ALIAS_SEND_TT` 控制消息。在 OTP 24 中引入。

还有一个函数 `dist_util:strict_order_flags/0` 返回所有标志（(bitwise or'ed together)），这些标志对应于需要通过 distribution channels 对数据进行严格校验的功能。（注：大概是奇偶校验）

13.3 已连接节点通信协议

从 ERTS 5.7.2 (OTP R13B) 开始，运行时系统会在握手阶段传递一个分发标志，以便在所有传递的消息上使用 **distribution header**。在这种情况下，节点之间传递的消息具有以下格式：

	A	B	C	D
1	4	d	n	m
2	Length	DistributionHeader	ControlMessage	Message

Table 13.23: Format of Messages Passed between Nodes (as from ERTS 5.7.2 (OTP R13B))

Length

等于 $d + n + m$ 。

Distribution Header

Distribution header describing the atom cache and fragmented distribution messages.

Control Message

使用 Erlang 的外部格式传递的元组。

Message

使用 “!”（注：erlang 进程通信符，同 `send` 函数）发送到另一个 node 的消息 或使用外部术语格式的 EXIT、EXIT2 或 DOWN 信号的原因。

注意 **the version number is omitted from the terms that follow a distribution header** .

ERTS 版本早于 5.7.2 (OTP R13B) 的节点不会传递启用分发标头的分发标志。在这种情况下，节点之间传递的消息具有以下格式：

	A	B	C	D
1	4	1	n	m
2	Length	Type	ControlMessage	Message

Table 13.24: Format of Messages Passed between Nodes (before ERTS 5.7.2 (OTP R13B))

Length

等于 $1 + n + m$ 。

Type

等于 112 (pass through).

ControlMessage

使用 Erlang 的外部格式传递的元组。

Message

使用 “!” 发送到另一个节点的消息（外部格式）。请注意，Message 仅与 编码发送 (!') 的 ControlMessage 组合传递。

ControlMessage 是一个元组，其中第一个元素指示它的分布式操作编码：

LINK

{1, FromPid, ToPid}

该信号由 FromPid 发送，以便在 FromPid 和 ToPid 之间创建链接。

SEND

{2, Unused, ToPid}

Followed by Message.

Unused is kept for backward compatibility.

EXIT


{3, FromPid, ToPid, Reason}

This signal is sent when a link has been broken

UNLINK (**deprecated**)

{4, FromPid, ToPid}

当使用 **old link protocol** 时，该信号由 FromPid 发送，以便删除 FromPid 和 ToPid 之间的链接。

 此信号已被弃用，并且在 OTP 26 中不受支持。有关更多信息，请参阅 [new link protocol](#)

NODE_LINK

{5}

REG_SEND

{6, FromPid, Unused, ToName}

Followed by Message.

保留未使用是为了向后兼容。

GROUP_LEADER

{7, FromPid, ToPid}

EXIT2

{8, FromPid, ToPid, Reason}

这个信号是通过调用 `erlang:exit/2` 来发送的

SEND_TT

{12, Unused, ToPid, TraceToken}

Followed by Message.

保留未使用是为了向后兼容。

EXIT_TT

{13, FromPid, ToPid, TraceToken, Reason}

REG_SEND_TT

{16, FromPid, Unused, ToName, TraceToken}

Followed by Message.

保留未使用是为了向后兼容。

EXIT2_TT

{18, FromPid, ToPid, TraceToken, Reason}

MONITOR_P

{19, FromPid, ToProc, Ref}, where FromPid = monitoring process and ToProc = monitored process pid or name (atom)

DEMONITOR_P

{20, FromPid, ToProc, Ref}, where FromPid = monitoring process and ToProc = monitored process pid or name (atom)

We include FromPid just in case we want to trace this.

MONITOR_P_EXIT

{21, FromProc, ToPid, Ref, Reason}, where FromProc = monitored process pid or name (atom), ToPid = monitoring process, and Reason = exit reason for the monitored process

Erlang/OTP 21 的新 Ctrl 消息

SEND_SENDER

{22, FromPid, ToPid}

Followed by Message.

此控制消息替代SEND控制消息，并将 在连接建立握手中协商 分发标志DFLAG_SEND_SENDER时发送

💡 在建立连接之前编码的消息仍然可以使用SEND控制消息。但是，一旦发送了SEND_SENDER或SEND_SENDER_TT 控制消息， 将不再在连接上以相同方向发送 SEND控制消息。

SEND_SENDER_TT

{23, FromPid, ToPid, TraceToken}

Followed by Message.

此控制消息替代SEND_TT控制消息，并将 在连接建立握手中协商 分发标志DFLAG_SEND_SENDE R时发送。

💡 在建立连接之前编码的消息仍然可以使用SEND_TT控制消息。但是，一旦发送了SEND_SENDER或SEND_SENDER_TT 控制消息， 将不再在连接上以相同方向发送 SEND_TT控制消息。

Erlang/OTP 22 的新 Ctrl 消息

💡 在建立连接之前编码的消息可能仍使用非 PAYLOAD 变体。但是，一旦发送了 PAYLOAD 控制消息，就不会再在连接上以相同方向发送非 PAYLOAD 控制消息。

PAYLOAD_EXIT

{24, FromPid, ToPid}

Followed by Reason.

此控制消息替代EXIT控制消息，并将 在连接建立握手中协商 分发标志DFLAG_EXIT_PAYLOAD

PAYLOAD_EXIT_TT

{25, FromPid, ToPid, TraceToken}

Followed by Reason.

此控制消息替换EXIT2控制消息，并将在连接建立握手中协商分发标志DFLAG_EXIT_PAYLOAD时发送

PAYLOAD_EXIT2

{26, FromPid, ToPid}

Followed by Reason.

此控制消息替换EXIT2控制消息，并将在连接建立握手中协商分发标志DFLAG_EXIT_PAYLOAD时发送

PAYLOAD_EXIT2_TT

{27, FromPid, ToPid, TraceToken}

Followed by Reason.

此控制消息替代EXIT2_TT控制消息，并将在连接建立握手中协商分发标志DFLAG_EXIT_PAYLOAD时发送

PAYLOAD_MONITOR_P_EXIT

{28, FromProc, ToPid, Ref}

Followed by Reason.

此控制消息替换MONITOR_P_EXIT控制消息，并将在连接建立握手中协商分发标志DFLAG_EXIT_PAYLOAD时发送

Erlang/OTP 23 的新 Ctrl 消息

SPAWN_REQUEST

{29, ReqId, From, GroupLeader, {Module, Function, Arity}, OptList}

Followed by ArgList.

该信号由 `spawn_request()` BIF 发送。

ReqId :: reference()

Request identifier。也用作monitor reference，以防monitor option已过时。

From :: pid()

发出请求的进程的进程标识符。也就是要成为父进程。

GroupLeader :: pid()

新创建进程的group leader进程标识符。

{Module :: atom(), Function :: atom(), Arity :: integer() >= 0}

新流程的入口点。

OptList :: [term()]

。 A proper list of spawn options to use when spawning.

ArgList :: [term()]

A proper list of arguments to use in the call to the entry point.

Only supported when the **DFLAG_SPAWN distribution flag** has been passed.

SPAWN_REQUEST_TT

{30, ReqId, From, GroupLeader, {Module, Function, Arity}, OptList, Token}

Followed by ArgList.

同 **SPAWN_REQUEST**, but also with a sequential trace Token.

仅当 **DFLAG_SPAWN distribution flag** has been passed.

SPAWN_REPLY

{31, ReqId, To, Flags, Result}

该信号用来回复**SPAWN_REQUEST**信号。

ReqId :: reference()

Request identifier。也用作monitor reference，以防monitor option已过时。

To :: pid()

发出生成请求的进程的进程标识符。

Flags :: integer() >= 0

位标志字段 bitwise or:ed together. 目前定义了以下标志：

1

在Result所在的节点上建立了To和Result之间的链接。

2

在Result所在的节点上设置了一个从To到Result的monitor。

Result :: pid() | atom()

操作的结果。如果Result是进程标识符，则操作成功，进程标识符是新创建的进程的标识符。
如果Result 是一个原子，则操作失败并且原子识别失败原因。

仅当 **DFLAG_SPAWN distribution flag** has been passed才支持。（has been passed啥意思？ ? ）

SPAWN_REPLY_TT

{32, ReqId, To, Flags, Result, Token}

与**SPAWN_REPLY**相同, but also with a sequential trace Token.

仅当 **DFLAG_SPAWN distribution flag** has been passed才支持.

UNLINK_ID

{35, Id, FromPid, ToPid}

该信号由FromPid发送，以删除FromPid和ToPid之间的链接。这个 unlink 信号取代了 **UNLINK** 信号。除了发送者和接收者的进程标识符，UNLINK_ID信号还包含一个整数标识符 Id。Id的有效范围是 [1, (1 bsl 64) - 1]。ID将由接收者在 **UNLINK_ID_ACK** 信号中传回给发送者。Id必须唯一标识来自 FromPid 的所有尚未确认的UNLINK_ID信号中的UNLINK_ID 信号到ToPid。

该信号仅在 使用 **new link protocol** 协商 **DFLAG_UNLINK_ID distribution flag** 时使用

UNLINK_ID_ACK

{36, Id, FromPid, ToPid}

取消链接确认信号。该信号作为收到 **UNLINK_ID** 信号的确认发送。Id元素应与UNLINK_ID信号中的Id相同。FromPid 标识UNLINK_ID_ACK信号 的发送者， ToPid标识UNLINK_ID信号的发送者。

该信号仅在 使用 **new link protocol** 协商 **DFLAG_UNLINK_ID distribution flag** 时使用

Erlang/OTP 24 的新 Ctrl 消息

ALIAS_SEND

{33, FromPid, Alias}

Followed by Message.

此控制消息在将消息Message发送 到由进程别名Alias标识的进程时使用。可以处理此控制消息的节点 在连接建立握手中设置 distribution flag **DFLAG_ALIAS** 。

ALIAS_SEND_TT

{34, FromPid, Alias, Token}

Followed by Message.

Same as **ALIAS_SEND**, but also with a sequential trace Token.

Link Protocol

New Link Protocol

The new link protocol will be used when both nodes flag that they understand it using the **DFLAG_UNLINK_ID distribution flag**. If one of the nodes does not understand the new link protocol, the **old link protocol** will be used as a fallback.

The new link protocol introduces two new signals, **UNLINK_ID** and **UNLINK_ID_ACK**, which replace the old **UNLINK** signal. The old **LINK** signal is still sent in order to set up a link, but handled differently upon reception.

In order to set up a link, a LINK signal is sent, from the process initiating the operation, to the process that it wants to link to. In order to remove a link, an UNLINK_ID signal is sent, from the process initiating the operation, to the linked process. The receiver of an UNLINK_ID signal responds with an UNLINK_ID_ACK signal. Upon reception of an UNLINK_ID signal, the corresponding UNLINK_ID_ACK signal **must** be sent before any other signals are sent to the sender of the UNLINK_ID signal. Together with **the signal ordering guarantee** of Erlang this makes it possible for the sender of the UNLINK_ID signal to know the order of other signals which is essential for the protocol. The UNLINK_ID_ACK signal should contain the same Id as the Id contained in the UNLINK_ID signal being acknowledged.

Processes also need to maintain process local information about links. The state of this process local information is changed when the signals above are sent and received. This process local information also determines if a signal should be sent when a process calls `link/1` or `unlink/1`. A LINK signal is only sent if there does not currently exist an active link between the processes according to the process local information and an UNLINK_ID signal is only sent if there currently exists an active link between the processes according to the process local information.

The process local information about a link contains:

Pid Process identifier of the linked process. **Active Flag** If set, the link is active and the process will react on **incoming exit signals** issued due to the link. If not set, the link is inactive and incoming exit signals, issued due to the link, will be ignored. That is, the processes are considered as **not** linked. **Unlink Id** Identifier of an outstanding unlink operation. That is, an unlink operation that has not yet been acknowledged. This information is only used when the active flag is not set.

A process is only considered linked to another process if it has process local information about the link containing the process identifier of the other process and with the active flag set.

The process local information about a link is updated as follows:

A LINK signal is sent Link information is created if not already existing. The active flag is set, and unlink id is cleared. That is, if we had an outstanding unlink operation we will ignore the result of that operation and enable the link. **A LINK signal is received** If no link information already exists, it is created, the active flag is set and unlink id is cleared. If the link information already exists, the signal is silently ignored, regardless of whether the active flag is set or not. That is, if we have an outstanding unlink operation we will **not** activate the link. In this scenario, the sender of the LINK signal has not yet sent an UNLINK_ID_ACK signal corresponding to our UNLINK_ID signal which means that it will receive our UNLINK_ID signal after it sent its LINK signal. This in turn means that both processes in the end will agree that there is no link between them. **An UNLINK_ID signal is sent** Link information already exists and the active flag is set (otherwise the signal would not be sent). The active flag is unset, and the unlink id of the signal is saved in the link information. **An UNLINK_ID signal is received** If the active flag is set, information about the link is removed. If the active flag is not set (that is, we have an outstanding unlink operation), the

information about the link is left unchanged. **An UNLINK_ID_ACK signal is sent** This is done when an UNLINK_ID signal is received and causes no further changes of the link information. **An UNLINK_ID_ACK signal is received** If information about the link exists, the active flag is not set, and the unlink id in the link information equals the Id in the signal, the link information is removed; otherwise, the signal is ignored.

When a process receives an exit signal due to a link, the process will first react to the exit signal if the link is active and then remove the process local information about the link.

In case the connection is lost between two nodes, exit signals with exit reason noconnection are sent to all processes with links over the connection. This will cause all process local information about links over the connection to be removed.

Exactly the same link protocol is also used internally on an Erlang node. The signals however have different formats since they do not have to be sent over the wire.

Old Link Protocol

The old link protocol utilize two signals [LINK](#), and [UNLINK](#). The LINK signal informs the other process that a link should be set up, and the UNLINK signal informs the other process that a link should be removed. This protocol is however a bit too naive. If both processes operate on the link simultaneously, the link may end up in an inconsistent state where one process thinks it is linked while the other thinks it is not linked.

This protocol is deprecated and support for it will be removed in OTP 26. Until then, it will be used as fallback when communicating with old nodes that do not understand the [new link protocol](#).