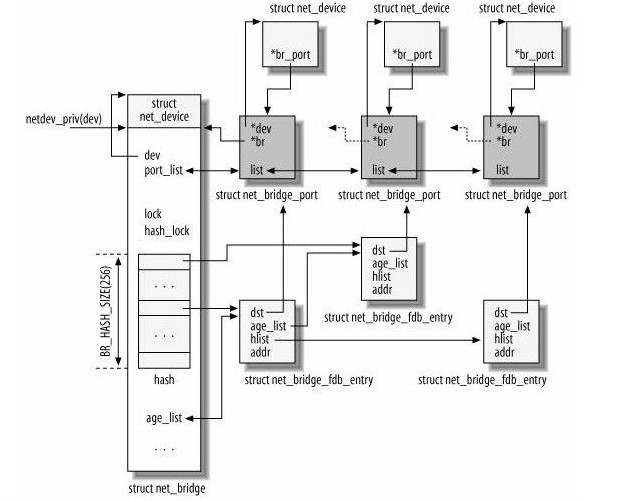
**数据结构**

1. // 转发数据表项
2. **struct** **net\_bridge\_fdb\_entry** {
3. /\*链接到net\_bridge的hash[BR\_HASH\_SIZE]\*/
4. **struct** hlist\_node       hlist;
5. /\*指向网桥端口\*/
6. **struct** **net\_bridge\_port**      \***dst**;
8. /\*mac地址\*/
9. **mac\_addr**            **addr**;
10. \_\_u16               vlan\_id;    //MAC属于哪个VLAN？
11. /\*mac地址是否为本地地址，1为是\*/
12. unsigned **char**           is\_local:1,
13. /\*mac地址为静态的，表示mac地址不会过期，本地地址都是静态的。\*/
14. is\_static:1,
15. added\_by\_user:1,    //用户配置
16. added\_by\_external\_learn:1,  //外部学习
17. offloaded:1;
19. /\* write-heavy members should not affect lookups \*/
20. unsigned **long**           **updated** \_\_\_\_cacheline\_aligned\_in\_smp;
21. unsigned **long**           used;   //引用计数
23. **struct** rcu\_head         rcu;
24. };
26. // 桥端口
27. **struct** **net\_bridge\_port** {
28. /\*指向网桥端口所属的网桥设备\*/
29. **struct** net\_bridge       \*br;
30. /\*添加到网桥的设备\*/
31. **struct** net\_device       \***dev**;   //本端口所指向的物理网卡  // dev->br\_port = dev
32. /\*网桥端口列表，连接到net\_bridge->head\_list\*/
33. **struct** list\_head        list;
35. /\*flags参数很重要\*/
36. unsigned **long**           flags;
37. #ifdef CONFIG\_BRIDGE\_VLAN\_FILTERING
38. **struct** net\_bridge\_vlan\_group    \_\_rcu \*vlgrp;
39. #endif
41. /\* STP \*/
42. u8              priority;   /\*端口优先级\*/
43. u8              state;  /\*端口状态\*/
44. u16             port\_no;    //本端口在网桥中的编号，唯一 id
45. unsigned **char**           topology\_change\_ack;
46. unsigned **char**           config\_pending;
47. port\_id             port\_id;    /\*端口ID，由端口优先级和端口号组成\*/
48. port\_id             designated\_port;    //指定端口的端口ID
49. bridge\_id           designated\_root;    //根网桥的网桥ID
50. bridge\_id           designated\_bridge;  //发送当前BPDU包的网桥的ID
51. u32             path\_cost;              //此端口的路径花销
52. u32             designated\_cost;        //到根桥的链路花销
53. unsigned **long**           designated\_age;
55. //端口定时器，也就是stp控制超时的一些定时器列表.
56. **struct** timer\_list       forward\_delay\_timer;
57. **struct** timer\_list       hold\_timer;
58. **struct** timer\_list       message\_age\_timer;
59. **struct** kobject          kobj;
60. **struct** rcu\_head         rcu;
61. ……
62. };

65. // net bridge
66. **struct** **net\_bridge** {
67. spinlock\_t          lock;   // 自旋锁，在向 net\_bridge 中增加 port 节点或改变 net\_bridge 结构时使用
68. spinlock\_t          hash\_lock;      // 对 hash 转发库进行操作时需要使用该自旋锁
69. **struct** list\_head        **port\_list**;  // 网桥端口列表
70. **struct** net\_device       \***dev**;       // 网桥设备
71. **struct** pcpu\_sw\_netstats     \_\_percpu \*stats;
72. unsigned **long**           options;
73. /\* These fields are accessed on each packet \*/
74. #ifdef CONFIG\_BRIDGE\_VLAN\_FILTERING
75. \_\_be16              vlan\_proto;
76. u16             default\_pvid;
77. **struct** net\_bridge\_vlan\_group    \_\_rcu \*vlgrp;
78. #endif
80. **struct** hlist\_head **hash[BR\_HASH\_SIZE]**;
81. #if IS\_ENABLED(CONFIG\_BRIDGE\_NETFILTER)
82. **union** {
83. **struct** rtable       fake\_rtable;
84. **struct** rt6\_info     fake\_rt6\_info;
85. };
86. #endif
87. u16             group\_fwd\_mask;
88. u16             group\_fwd\_mask\_required;
90. /\* STP \*/
91. bridge\_id           designated\_root;
92. bridge\_id           bridge\_id;
93. unsigned **char**           topology\_change;
94. unsigned **char**           topology\_change\_detected;
95. u16             root\_port;  // 根端口的端口号
97. /\* 网桥定时器 \*/
98. unsigned **long**           max\_age;
99. unsigned **long**           hello\_time;
100. unsigned **long**           forward\_delay;
101. unsigned **long**           ageing\_time;
103. /\* 本地配置的网桥定时器 \*/
104. unsigned **long**           bridge\_max\_age;
105. unsigned **long**           bridge\_hello\_time;
106. unsigned **long**           bridge\_forward\_delay;
107. unsigned **long**           bridge\_ageing\_time;
108. u32             root\_path\_cost;
110. u8              group\_addr[ETH\_ALEN];
112. **enum** {
113. BR\_NO\_STP,      /\* no spanning tree \*/
114. BR\_KERNEL\_STP,      /\* old STP in kernel \*/
115. BR\_USER\_STP,        /\* new RSTP in userspace \*/
116. } stp\_enabled;
118. ……
119. /\* 网桥定时器 \*/
120. **struct** timer\_list       hello\_timer;
121. **struct** timer\_list       tcn\_timer;
122. **struct** timer\_list       topology\_change\_timer;
123. **struct** delayed\_work     gc\_work;
124. **struct** kobject          \*ifobj;
125. u32             auto\_cnt;
126. ……
127. };



**功能函数**

1. // 所有网桥通信的数据包都会进入到这里，网桥处理函数
2. rx\_handler\_result\_t **br\_handle\_frame**(**struct** sk\_buff \*\*pskb)
3. {
5. /\*调用NF\_BR\_PRE\_ROUTING处钩子函数，结束后进入br\_handle\_frame\_finish函数\*/
6. NF\_HOOK(NFPROTO\_BRIDGE, NF\_BR\_PRE\_ROUTING,
7. dev\_net(skb->dev), NULL, skb, skb->dev, NULL,
8. **br\_handle\_frame\_finish**);
9. }
10. **int** **br\_handle\_frame\_finish**(**struct** net \*net, **struct** sock \*sk, **struct** sk\_buff \*skb)
11. {
12. **struct** net\_bridge\_port \*p = br\_port\_get\_rcu(skb->dev);
13. /\*如果网桥端口标志有BR\_LEARNING,则更新fdb表。
14. 一般新建网桥端口p->flags=BR\_LEARNING| BR\_FLOOD | BR\_MCAST\_FLOOD | BR\_BCAST\_FLOOD\*/
15. **if** (p->flags & BR\_LEARNING)
16. **br\_fdb\_update**(br, p, eth\_hdr(skb)->h\_source, vid, **false**);
18. //根据目的MAC地址查找fdb表，看是否有对应的表项
19. dst = **br\_fdb\_find\_rcu**(br, eth\_hdr(skb)->h\_dest, vid);
21. //如果找到目的MAC对应的表项
22. **if** (dst) {  /\* 目的mac对应的fdb项如果存在，不是广播或者多播的情况下,判断是否本地地址，如果是本地地址，调用br\_pass\_frame\_up发往本地。 否则调用br\_forward进行数据包转发\*/
24. // 目的地址是本机（网桥mac）
25. // 送入上层处理
26. **if** (dst->is\_local)
27. **return** **br\_pass\_frame\_up**(skb);
29. /\* 目的地址不是本机，非网桥mac \*/
30. // 传入的第一个参数dst->dst 即为要转发的目的端口
31. **br\_forward**(dst->dst, skb, local\_rcv, **false**); /\*转发表中存在并且不是本地的，即需要转发到其它端口dst->dst\*/
32. } **else** {
33. //进行广播或者组播洪泛
34. **if** (!mcast\_hit)
35. **br\_flood**(br, skb, pkt\_type, local\_rcv, **false**);
36. **else**
37. **br\_multicast\_flood**(mdst, skb, local\_rcv, **false**);
38. }
40. }
41. **static** **int** **br\_pass\_frame\_up**(**struct** sk\_buff \*skb)
42. {
44. //统计网桥设备上的收包流量数据
45. u64\_stats\_update\_begin(&brstats->syncp);
46. brstats->rx\_packets++;
47. brstats->rx\_bytes += skb->len;
48. u64\_stats\_update\_end(&brstats->syncp);

51. //记录数据包的收包网络设备
52. indev = skb->dev;
53. //将数据包的收包设备改为网桥设备
54. //当再次进入\_\_netif\_receive\_skb\_core时就不会再次进入桥处理了，因为网桥上没有注册 rx\_handler 函数
55. **skb->dev = brdev;**
56. skb = br\_handle\_vlan(br, NULL, vg, skb);
57. **if** (!skb)
58. **return** NET\_RX\_DROP;
60. //调用NF\_BR\_LOCAL\_IN处钩子函数，最后调用br\_netif\_receive\_skb函数。
61. //再次进入netif\_receive\_skb，由于skb-dev被设置成了bridge，而bridge设备的rx\_handler函数是没有被设置的，所以就不会再次进入bridge逻辑，而直接进入了主机上层协议栈。
62. **return** NF\_HOOK(NFPROTO\_BRIDGE, NF\_BR\_LOCAL\_IN,
63. dev\_net(indev), NULL, skb, indev, NULL,
64. **br\_netif\_receive\_skb**);
65. }
66. **static** **int**
67. **br\_netif\_receive\_skb**(**struct** net \*net, **struct** sock \*sk, **struct** sk\_buff \*skb)
68. {
69. br\_drop\_fake\_rtable(skb);
70. **return** **netif\_receive\_skb**(skb);
71. }
72. /\*\*
73. \* br\_forward - forward a packet to a specific port
74. \* **@to: destination port**
75. \* @skb: packet being forwarded
76. \* @local\_rcv: packet will be received locally after forwarding
77. \* @local\_orig: packet is locally originated
78. \*/
79. **void** **br\_forward**(**const** **struct** net\_bridge\_port \***to**,
80. **struct** sk\_buff \*skb, **bool** local\_rcv, **bool** local\_orig)
81. {
82. **\_\_br\_forward**(to, skb, local\_orig);
83. }
84. **static** **void** **\_\_br\_forward**(**const** **struct** net\_bridge\_port \*to,
85. **struct** sk\_buff \*skb, **bool** local\_orig)
86. {
87. NF\_HOOK(NFPROTO\_BRIDGE, br\_hook,
88. net, NULL, skb, indev, skb->dev,
89. **br\_forward\_finish**);
90. }
91. **int** br\_forward\_finish(**struct** net \*net, **struct** sock \*sk, **struct** sk\_buff \*skb)
92. {
93. skb->tstamp = 0;
94. **return** NF\_HOOK(NFPROTO\_BRIDGE, NF\_BR\_POST\_ROUTING,
95. net, sk, skb, NULL, skb->dev,
96. **br\_dev\_queue\_push\_xmit**);  // 🡪 dev\_queue\_xmit() //当所有的信息（出口设备、下一跳地址、链路层地址）都准备好了之后，就会调用dev\_queue\_xmit()，该函数时设备驱动程序执行传输的接口。也就是所有的数据包在填充完成后，最终发送时，都会调用该函数。
98. }
99. **void** **br\_flood**(**struct** net\_bridge \*br, **struct** sk\_buff \*skb,
100. **enum** br\_pkt\_type pkt\_type, **bool** local\_rcv, **bool** local\_orig)
101. {
102. list\_**for\_each**\_entry\_rcu(p, &**br->port\_list**, list) {
103. /\* Do not flood unicast traffic to ports that turn it off, nor
104. \* other traffic if flood off, except for traffic we originate
105. \*/
106. **switch** (pkt\_type) {
107. **case** BR\_PKT\_UNICAST:
108. **if** (!(p->flags & BR\_FLOOD))
109. **continue**;
110. **break**;
111. **case** BR\_PKT\_MULTICAST:
112. **if** (!(p->flags & BR\_MCAST\_FLOOD) && skb->dev != br->dev)
113. **continue**;
114. **break**;
115. **case** BR\_PKT\_BROADCAST:
116. **if** (!(p->flags & BR\_BCAST\_FLOOD) && skb->dev != br->dev)
117. **continue**;
118. **break**;
119. }
121. /\* Do not flood to ports that enable proxy ARP \*/
122. **if** (p->flags & BR\_PROXYARP)
123. **continue**;
124. **if** ((p->flags & (BR\_PROXYARP\_WIFI | BR\_NEIGH\_SUPPRESS)) &&
125. BR\_INPUT\_SKB\_CB(skb)->proxyarp\_replied)
126. **continue**;
128. prev = **maybe\_deliver**(prev, p, skb, local\_orig);  // 🡪 \_\_br\_forward()
129. **if** (IS\_ERR(prev))
130. **goto** out;
131. **if** (prev == p)
132. br\_multicast\_count(p->br, p, skb, igmp\_type,
133. BR\_MCAST\_DIR\_TX);
134. }
136. }
137. **void** **br\_multicast\_flood**(**struct** net\_bridge\_mdb\_entry \*mdst,
138. **struct** sk\_buff \*skb,
139. **bool** local\_rcv, **bool** local\_orig)
140. {
141. **while (p || rp)**{
142. **struct** net\_bridge\_port \*port, \*lport, \*rport;
144. lport = p ? p->port : NULL;
145. rport = rp ? hlist\_entry(rp, **struct** net\_bridge\_port, rlist) :
146. NULL;
148. **if** ((unsigned **long**)lport > (unsigned **long**)rport) {
149. port = lport;
151. **if** (port->flags & BR\_MULTICAST\_TO\_UNICAST) {
152. **maybe\_deliver\_addr**(lport, skb, p->eth\_addr,  // 🡪 \_\_br\_forward()
153. local\_orig);
154. **goto** delivered;
155. }
156. } **else** {
157. port = rport;
158. }
160. }
162. }
163. **int** **br\_add\_bridge**(**struct** net \*net, **const** **char** \*name)
164. {
165. **struct** net\_device \*dev;
166. **int** res;
168. // 申请 net\_device 的内存、调用 br\_dev\_setup 对网桥设备进行初始化
169. dev = alloc\_netdev(**sizeof**(**struct** net\_bridge), name, NET\_NAME\_UNKNOWN,
170. br\_dev\_setup);
172. **if** (!dev)
173. **return** -ENOMEM;
175. dev\_net\_set(dev, net);
176. dev->rtnl\_link\_ops = &br\_link\_ops;
178. res = register\_netdev(dev); // 注册网桥设备
179. **if** (res)
180. free\_netdev(dev);
181. **return** res;
182. }
183. **int** **br\_del\_bridg**e(**struct** net \*net, **const** **char** \*name)
184. {
185. **struct** net\_device \*dev;
186. **int** ret = 0;
188. pr\_info("b2. brctl delbr <bridge>-br\_del\_bridge\n");
190. rtnl\_lock();
191. dev = \_\_dev\_get\_by\_name(net, name);
192. **if** (dev == NULL)
193. ret =  -ENXIO;  /\* Could not find device \*/
195. **else** **if** (!(dev->priv\_flags & IFF\_EBRIDGE)) {
196. /\* Attempt to delete non bridge device! \*/
197. ret = -EPERM;
198. }
200. **else** **if** (dev->flags & IFF\_UP) {
201. /\* Not shutdown yet. \*/
202. ret = -EBUSY;
203. }
205. **else**
206. br\_dev\_delete(dev, NULL);
208. rtnl\_unlock();
209. **return** ret;
210. }
211. **int** **br\_add\_if**(**struct** net\_bridge \*br, **struct** net\_device \*dev,
212. **struct** netlink\_ext\_ack \*extack)
213. {
215. // 回环，或者非以太网接口不可以加入bridge
216. **if** ((dev->flags & IFF\_LOOPBACK) ||
217. dev->type != ARPHRD\_ETHER || dev->addr\_len != ETH\_ALEN ||
218. !is\_valid\_ether\_addr(dev->dev\_addr) ||
219. netdev\_uses\_dsa(dev))
220. **return** -EINVAL;
222. //桥设备不能再加入桥
223. /\* No bridging of bridges \*/
224. **if** (dev->netdev\_ops->ndo\_start\_xmit == br\_dev\_xmit) {
225. NL\_SET\_ERR\_MSG(extack,
226. "Can not enslave a bridge to a bridge");
227. **return** -ELOOP;
228. }
230. /\* Device has master upper dev \*/
231. **if** (netdev\_master\_upper\_dev\_get(dev))
232. **return** -EBUSY;
234. /\* No bridging devices that dislike that (e.g. wireless) \*/
235. **if** (dev->priv\_flags & IFF\_DONT\_BRIDGE) {
236. NL\_SET\_ERR\_MSG(extack,
237. "Device does not allow enslaving to a bridge");
238. **return** -EOPNOTSUPP;
239. }
241. // 分配一个新网桥端口并对其初始化
242. p = new\_nbp(br, dev);
243. **if** (IS\_ERR(p))
244. **return** PTR\_ERR(p);
246. // 调用设备通知链，告诉网络有这样一个设备
247. call\_netdevice\_notifiers(NETDEV\_JOIN, dev);
249. // 注册设备接收函数
250. err = **netdev\_rx\_handler\_register**(dev, **br\_handle\_frame**, p);
251. **if** (err)
252. **goto** err4;
254. /\*更新桥上的端口数，进一步将其设为**混杂模式**\*/
255. nbp\_update\_port\_count(br);
257. // 添加到fdb里 mac--port
258. **if** (br\_fdb\_insert(br, p, dev->dev\_addr, 0))
259. netdev\_err(dev, "failed insert local address bridge forwarding table\n");
261. // 设备是否启动，桥是否启动，设备上是否有载波信号
262. **if** (netif\_running(dev) && netif\_oper\_up(dev) &&
263. (br->dev->flags & IFF\_UP))
264. // 启动网桥端口，这里port状态为转发
265. br\_stp\_enable\_port(p);

268. // 设置mtu
269. dev\_set\_mtu(br->dev, br\_min\_mtu(br));
270. }
271. **int** **br\_del\_if**(**struct** net\_bridge \*br, **struct** net\_device \*dev)
272. {
273. **struct** net\_bridge\_port \*p;
274. **bool** changed\_addr;
276. p = br\_port\_get\_rtnl(dev);
277. **if** (!p || p->br != br)
278. **return** -EINVAL;
280. /\* Since more than one interface can be attached to a bridge,
281. \* there still maybe an alternate path for netconsole to use;
282. \* therefore there is no reason for a NETDEV\_RELEASE event.
283. \*/
284. del\_nbp(p);
286. dev\_set\_mtu(br->dev, br\_min\_mtu(br));
287. br\_set\_gso\_limits(br);
289. spin\_lock\_bh(&br->lock);
290. changed\_addr = br\_stp\_recalculate\_bridge\_id(br);
291. spin\_unlock\_bh(&br->lock);
293. **if** (changed\_addr)
294. call\_netdevice\_notifiers(NETDEV\_CHANGEADDR, br->dev);
296. netdev\_update\_features(br->dev);
298. **return** 0;
299. }
300. /\* 将skb传递到上层 \*/
301. **static** **int** **\_\_netif\_receive\_skb\_core**(**struct** sk\_buff \*skb, **bool** pfmemalloc)
302. {
304. /\* 记录收包时间，netdev\_tstamp\_prequeue为0，表示可能有包延迟 \*/
305. net\_timestamp\_check(!netdev\_tstamp\_prequeue, skb);
307. trace\_netif\_receive\_skb(skb);
309. /\* 记录收包设备 \*/
310. orig\_dev = skb->dev;

313. another\_round:
315. /\* 处理包数统计 \*/
316. \_\_this\_cpu\_inc(softnet\_data.processed);

319. /\* 如果有注册handler，那么调用，比如网桥模块 \*/
320. rx\_handler = rcu\_dereference(**skb->dev->rx\_handler**);
321. **if** (rx\_handler) {
322. **if** (pt\_prev) {
323. ret = deliver\_skb(skb, pt\_prev, orig\_dev);
324. pt\_prev = NULL;
325. }
326. **switch** (rx\_handler(&skb)) {
327. /\* 已处理，无需进一步处理 \*/
328. **case** RX\_HANDLER\_CONSUMED:
329. ret = NET\_RX\_SUCCESS;
330. **goto** out;
331. /\* 修改了skb->dev，在处理一次 \*/
332. **case** RX\_HANDLER\_ANOTHER:
333. **goto** another\_round;
334. /\* 精确传递到ptype->dev == skb->dev \*/
335. **case** RX\_HANDLER\_EXACT:
336. deliver\_exact = **true**;
337. /\* 正常传递即可 \*/
338. **case** RX\_HANDLER\_PASS:
339. **break**;
340. **default**:
341. BUG();
342. }
343. }

346. /\* 设置三层协议，下面提交都是按照三层协议提交的 \*/
347. type = skb->protocol;
349. /\* deliver only exact match when indicated \*/
350. /\* 未设置精确发送，则向未指定设备的指定协议全局发送一份 \*/
351. **if** (likely(!deliver\_exact)) {
352. deliver\_ptype\_list\_skb(skb, &pt\_prev, orig\_dev, type,
353. &ptype\_base[ntohs(type) &
354. PTYPE\_HASH\_MASK]);
355. }
357. /\* 指定设备的，向原设备上层传递  \*/
358. deliver\_ptype\_list\_skb(skb, &pt\_prev, orig\_dev, type,
359. &orig\_dev->ptype\_specific);
361. /\*  当前设备与原设备不同，向当前设备传递 \*/
362. **if** (unlikely(skb->dev != orig\_dev)) {
363. deliver\_ptype\_list\_skb(skb, &pt\_prev, orig\_dev, type,
364. &skb->dev->ptype\_specific);
365. }
367. **if** (pt\_prev) {
368. **if** (unlikely(skb\_orphan\_frags(skb, GFP\_ATOMIC)))
369. **goto** drop;
370. **else**
371. /\*
372. 使用pt\_prev这里就不需要deliver\_skb来inc应用数了
373. func执行内部会free，减少了一次skb\_free
374. \*/
375. /\* 传递到上层\*/
376. ret = pt\_prev->func(skb, skb->dev, pt\_prev, orig\_dev);
377. } **else** {
378. drop:
379. **if** (!deliver\_exact)
380. atomic\_long\_inc(&skb->dev->rx\_dropped);
381. **else**
382. atomic\_long\_inc(&skb->dev->rx\_nohandler);
383. kfree\_skb(skb);
384. /\* Jamal, now you will not able to escape explaining
385. \* me how you were going to use this. :-)
386. \*/
387. ret = NET\_RX\_DROP;
388. }
390. out:
391. **return** ret;
392. }