pol\_nrs

pol\_private

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

nrs\_tbf\_jobid

**nrs\_tbf\_jobid**

nrs\_tbf\_jobid

rule

rule

rule

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

nrs\_policy\_primary

**ptlrpc\_service\_part**

**next**

**prev**

**nrs\_policy\_queued**

ptlrpc\_nrs\_policy

**next**

**prev**

**nrs\_policy\_queued**

**next**

**prev**

**nrs\_policy\_queued**

ptlrpc\_nrs\_policy

ptlrpc\_nrs\_policy

struct ptlrpc\_nrs\_resource \*res\_parent;

struct ptlrpc\_nrs\_policy \*res\_policy;

ptlrpc\_nrs\_resource

1

th\_list

primary

nrs\_tbf\_head

th\_rule(默认的)

**ptlrpc\_nrs(HP)**

**ptlrpc\_nrs(REG)**

th\_binheap

|  |
| --- |
| Resources 会被嵌入到两种类型NRS entities:  1. 在NRS policies内部，在policy的私有数据域 ptlrpc\_nrs\_policy::pol\_private  2. 在各种策略作为prime-level scheduling entities的对象中  例如在基于client NID轮询或者类似顺序的策略中，每个唯一的client NID会有一个NRS source在基于后备文件系统对象的轮询中调度中，每个后备文件系统系统partaking对应着一个Resource  NRS resources 之间维持着一个parent-child关系，嵌入到policy instance中的是parent entity，组成了一个简单的资源层级关系.以后还可以有更复杂的关系  struct ptlrpc\_nrs\_resource {  struct ptlrpc\_nrs\_resource \*res\_parent;//如果是NULL,那么该对象就是嵌入到policy实例中的那个，也即top-level级别的对象  struct ptlrpc\_nrs\_policy \*res\_policy;//与之关联的policy  }  Resources维系的主要是policy，因为一个policy通常是在一个osc上设置的 |

nrs\_tbf\_head

Fallback

top\_level

th\_cli\_hash

nrs\_XXX\_client

ptlrpc\_nrs\_request

nrs\_tbf\_req

**next**

**prev**

**nrs\_policy\_queued**

NRS\_RES\_FALLBACK

NRS\_RES\_PRIMARY,

nr\_res\_ptrs[NRS\_RES\_MAX],==2

**next**

**prev**

**nrs\_policy\_queued**

key:lnet\_nid\_t nid;/jobid取决于rule 设置

nrs\_tbf\_bucket LRU

位置和rule中一些参数相关

ptlprc\_request

rq\_srv

sr\_nrq

nrs\_policy\_fallback

ptlrpc\_nrs\_request

nrs\_xxx\_req

tc\_list

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

* **next**

**prev**

**nrs\_policy\_list**

nrs\_XXX\_client

ptlrpc\_nrs\_request

nrs\_tbf\_req

**next**

**prev**

**nrs\_policy\_queued**

ptlrpc\_nrs\_request

nrs\_tbf\_req

**next**

**prev**

**nrs\_policy\_queued**

ptlrpc\_nrs\_request

nrs\_tbf\_req

**next**

**prev**

**nrs\_policy\_queued**

* **next**

**prev**

**nrs\_policy\_list**

nrs\_XXX\_net

nrs\_orr\_object

nrs\_XXX\_client

nrs\_XXX\_client

nrs\_tbf\_client被维持在一个hash结构中，其中的key是 req->rq\_peer.nid，是

typedef struct {

/\*\* node id \*/

lnet\_nid\_t nid;

/\*\* process id \*/

lnet\_pid\_t pid;

} lnet\_process\_id\_t;

这个结构体中的nid

enum {

NRS\_RES\_FALLBACK,

NRS\_RES\_PRIMARY,

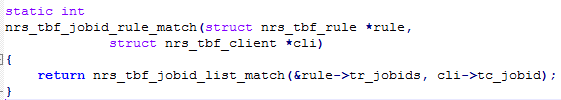
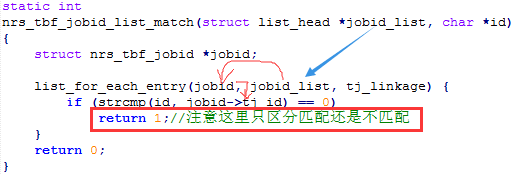
NRS\_RES\_MAX

};

struct ptlrpc\_nrs\_resource \*\*resp= memset(resp, 0, sizeof(resp[0]) \* NRS\_RES\_MAX);//NRS\_RES\_FALLBACK,NRS\_RES\_PRIMARY,NRS\_RES\_MAX

//NRS\_RES\_FALLBACK,NRS\_RES\_PRIMARY,NRS\_RES\_MAX分别为0,1,2

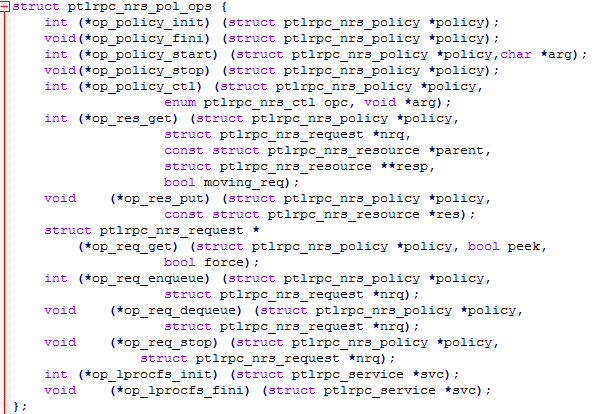
\*resp [0]也即\*resp [NRS\_RES\_FALLBACK], \*resp [1]也即\*resp [ NRS\_RES\_PRIMARY],



|  |
| --- |
| static int tbf\_jobid\_cache\_size = 8192;  CFS\_MODULE\_PARM(tbf\_jobid\_cache\_size, "i", int, 0644, "The size of jobid cache");  static int tbf\_rate = 10000;  CFS\_MODULE\_PARM(tbf\_rate, "i", int, 0644, "Default rate limit in RPCs/s");  static int tbf\_depth = 3;// tbf\_depth的意义  CFS\_MODULE\_PARM(tbf\_depth, "i", int, 0644, "How many tokens that a client can save up"); |

static int nrs\_tbf\_rule\_start(struct ptlrpc\_nrs\_policy \*policy, struct nrs\_tbf\_head \*head, struct nrs\_tbf\_cmd \*start)对rule进行初始化赋值操作

lnet\_nid\_t 、 jobid看规则是什么，其中维持cli hash表的key就是什么



truct ptlrpc\_nrs\_pol\_conf {

char nc\_name[NRS\_POL\_NAME\_MAX];

**const struct ptlrpc\_nrs\_pol\_ops \*nc\_ops;**

nrs\_pol\_desc\_compat\_t nc\_compat;

const char \*nc\_compat\_svc\_name;

struct module \*nc\_owner;

unsigned nc\_flags;

};

static int nrs\_policy\_register(struct ptlrpc\_nrs \*nrs,struct ptlrpc\_nrs\_pol\_desc \*desc)

{

rc = nrs\_policy\_init(policy);//有的是空操作

list\_add\_tail(&policy->pol\_list, &nrs->nrs\_policy\_list);

nrs->nrs\_num\_pols++;

if (policy->pol\_flags & PTLRPC\_NRS\_FL\_REG\_START)

rc = nrs\_policy\_start\_locked(policy, NULL);

}

**nrs.c**

int ptlrpc\_nrs\_policy\_register(struct ptlrpc\_nrs\_pol\_conf \*conf)

{

conf ===> struct ptlrpc\_nrs\_pol\_desc \*desc;

nrs = nrs\_svcpt2nrs(svcpt, hp);

nrs\_policy\_register(nrs, desc);

}

**nrs.c**

int ptlrpc\_init()

{

//各种初始化调用操作

rc = ptlrpc\_nrs\_init();

}

**ptlrpc\_module.c**

int ptlrpc\_nrs\_init(void)

{

rc = ptlrpc\_nrs\_policy\_register(&nrs\_conf\_tbf);

//其它policy注册及其相关的操作

}

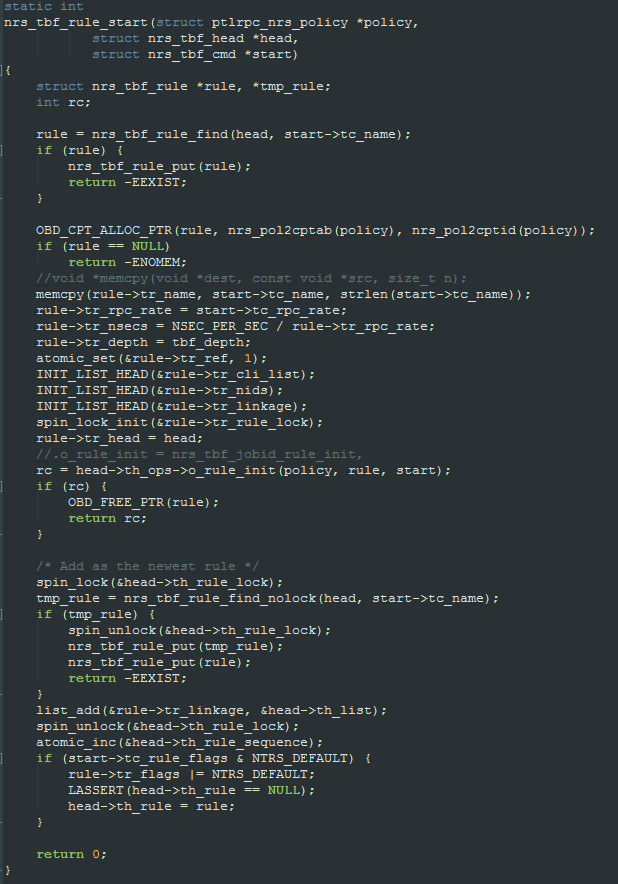
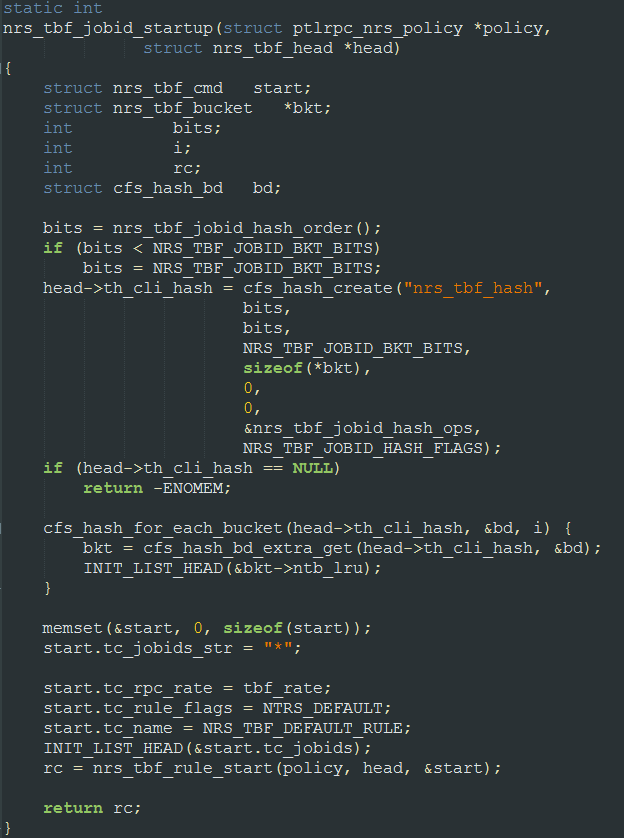
**nrs.c**

在static void nrs\_tbf\_jobid\_cli\_put(struct nrs\_tbf\_head \*head, struct nrs\_tbf\_client \*cli)函数中，cli被添加到hash bucket的底部，从而实现了LRU算法

#define NRS\_TBF\_JOBID\_BKT\_BITS 10

struct nrs\_tbf\_head \*head;

在nrs\_tbf\_start函数中 policy->pol\_private = head;



struct ptlrpc\_service \*ptlrpc\_register\_service(struct ptlrpc\_service\_conf \*conf, struct proc\_dir\_entry \*proc\_entry)

int ptlrpc\_start\_threads(struct ptlrpc\_service \*svc)

int ptlrpc\_start\_thread(struct ptlrpc\_service\_part \*svcpt, int wait)

static int ptlrpc\_main(void \*arg)

static int ptlrpc\_server\_handle\_request(struct ptlrpc\_service\_part \*svcpt, struct ptlrpc\_thread \*thread)

static struct ptlrpc\_request \*ptlrpc\_server\_request\_get(struct ptlrpc\_service\_part \*svcpt, bool force)

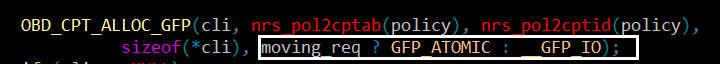
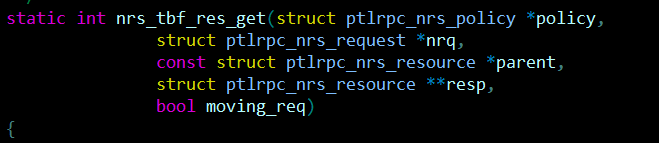
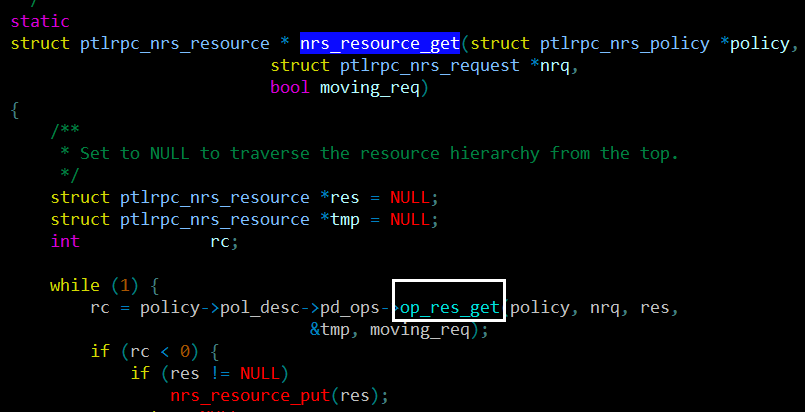
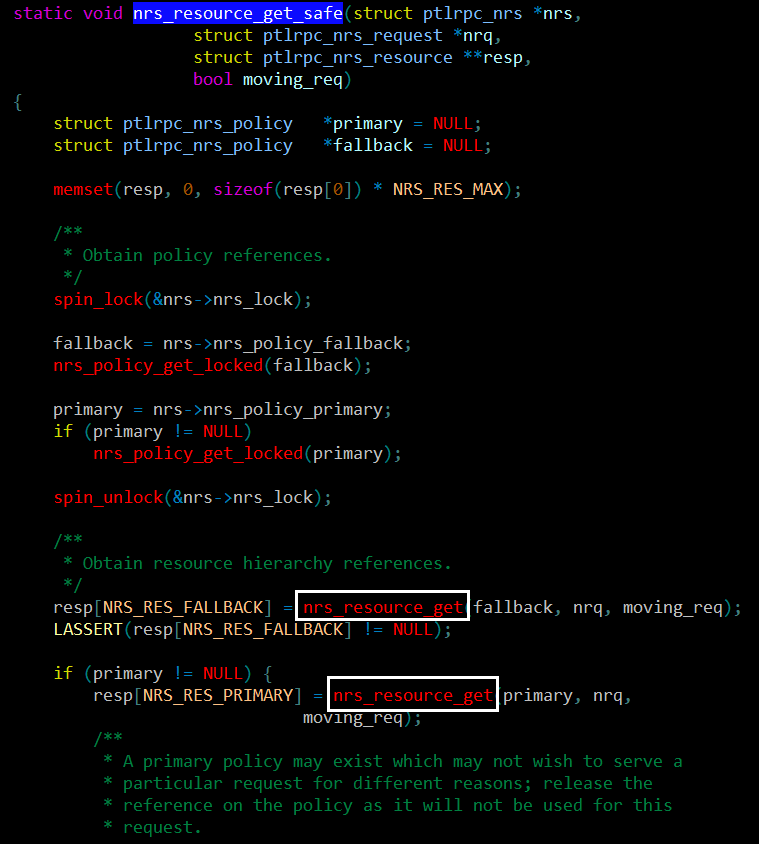
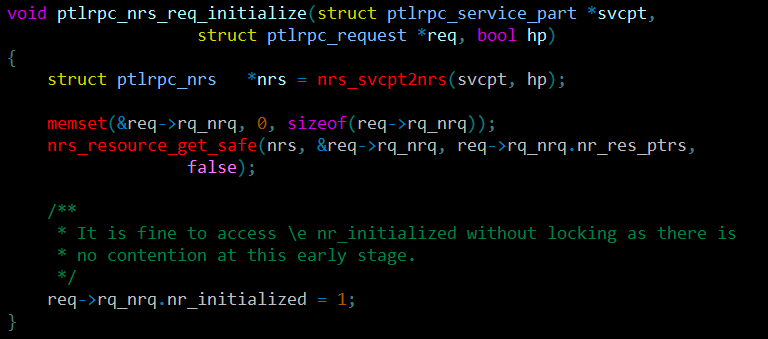
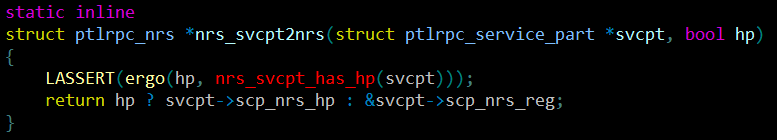
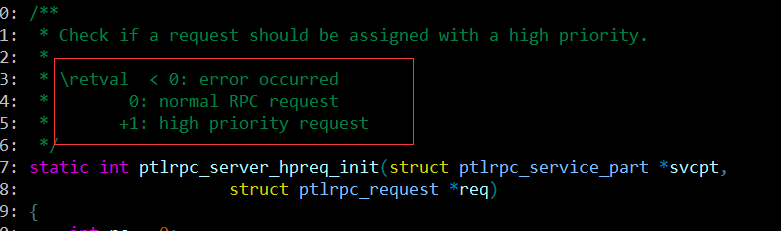
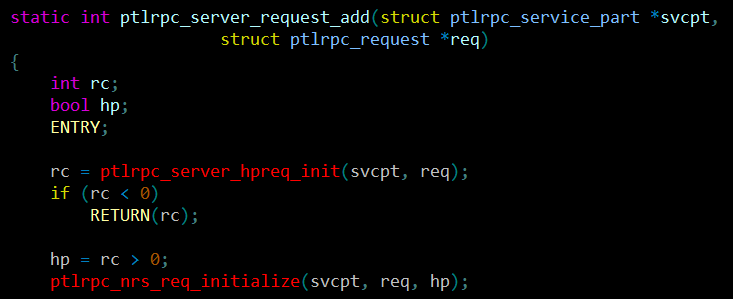
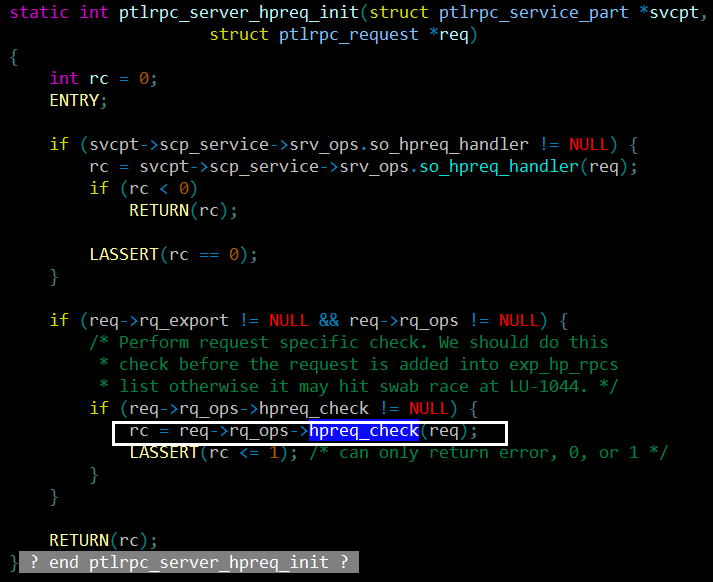
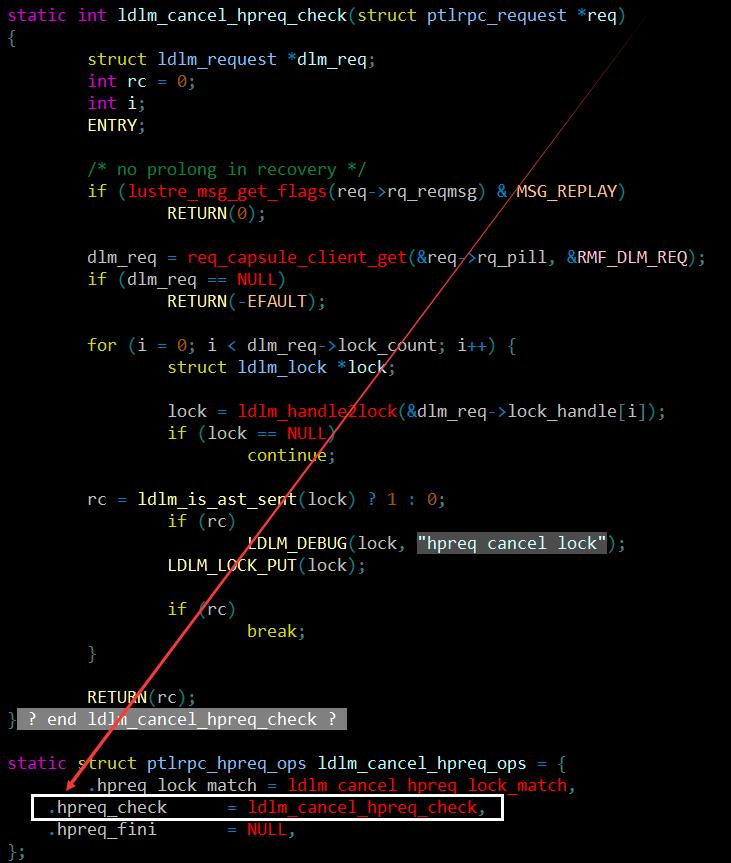
static inline struct ptlrpc\_request \*ptlrpc\_nrs\_req\_get\_nolock(struct ptlrpc\_service\_part \*svcpt, bool hp, bool force)

{

return ptlrpc\_nrs\_req\_get\_nolock0(svcpt, hp, false, force);

}

static inline struct ptlrpc\_nrs\_request \* nrs\_request\_get(struct ptlrpc\_nrs\_policy \*policy, bool peek, bool force)



**op\_res\_get**

**参见下面**

策略注册nrs\_policy\_register的时候调用nrs\_policy\_start\_locked，给其赋值

**在上面ptlrpc\_server\_request\_add函数中，添加请求是首先要通过相关函数确定这个请求是属于hp还是rg类型的。将相关函数返回的hp/rg类型作为一个参数传进函数ptlrpc\_nrs\_req\_initialize中，着直接决定了在nrs\_svcpt2nrs中得到的是什么类型的nrs**

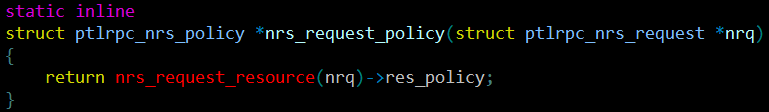
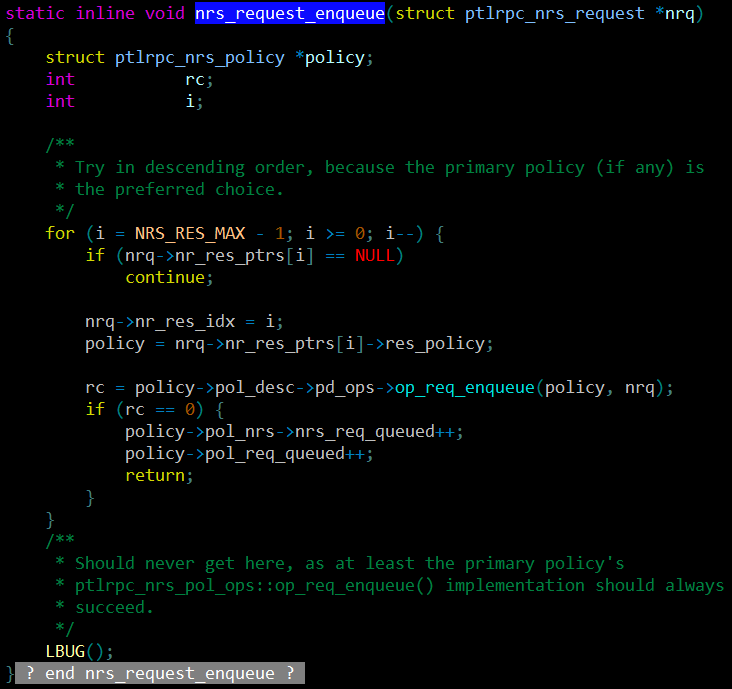
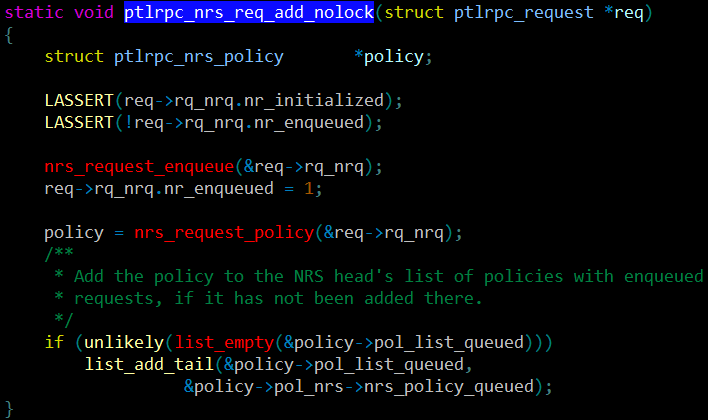
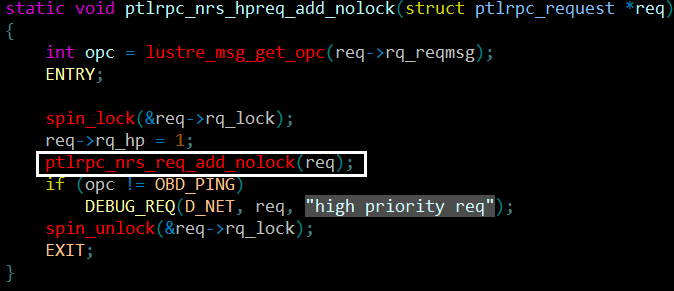
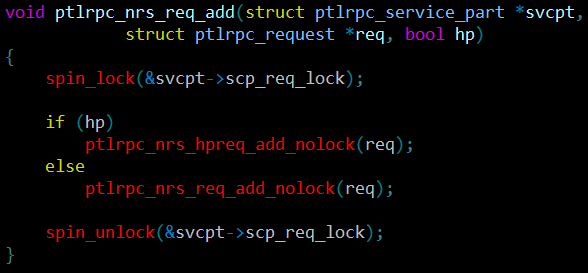
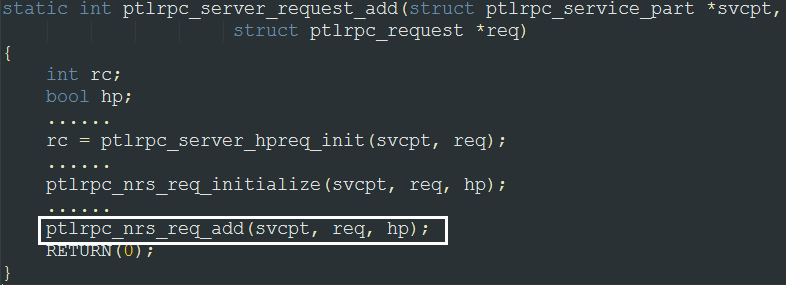
**而到底是什么请求：https://jira.hpdd.intel.com/browse/LU-398**

**Request Priority**

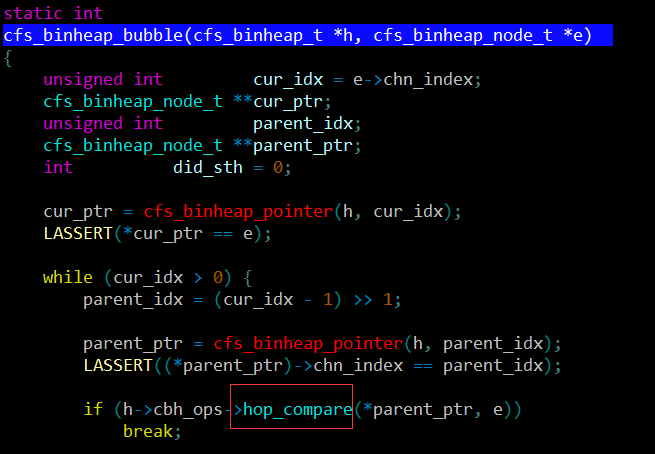
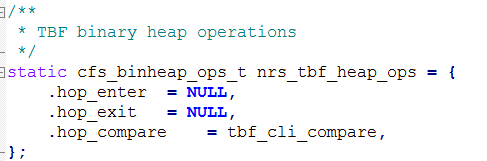
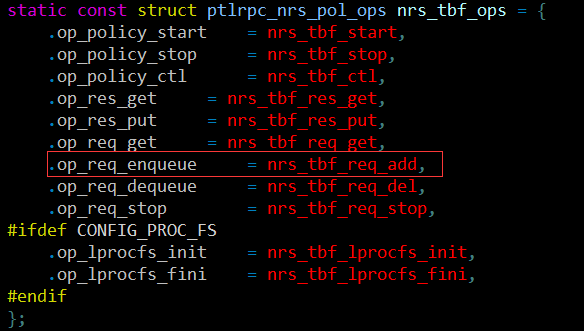
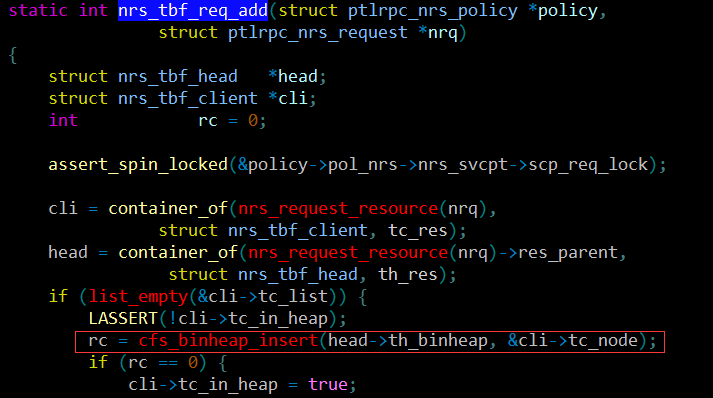
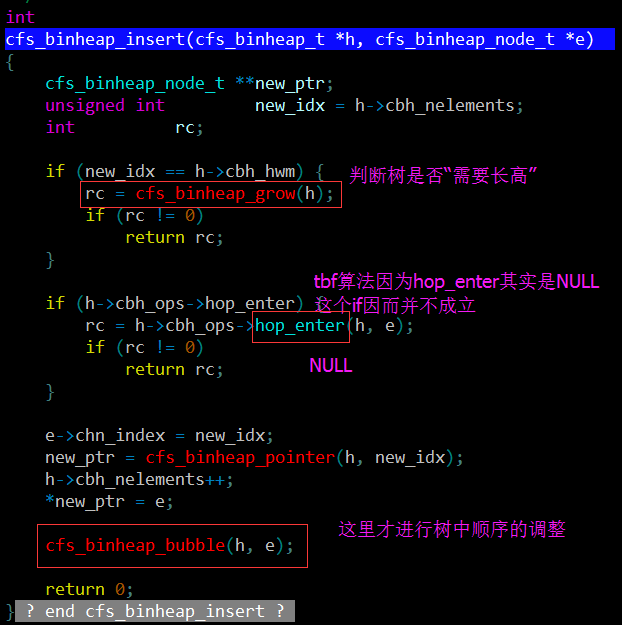
**Request priorities enable important requests to be serviced with lower latency - e.g. writes required to clean a cache on a locking conflict. Note that high priority requests must not break any POP requirements.**

**ptlrpc\_nrs\_req\_add**

**参见下面**



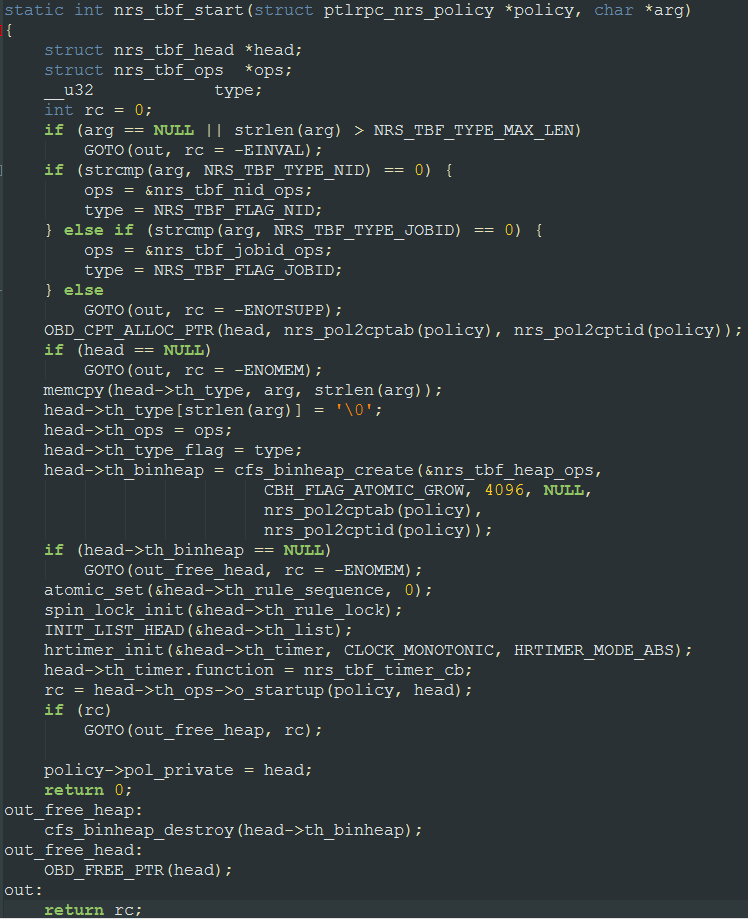
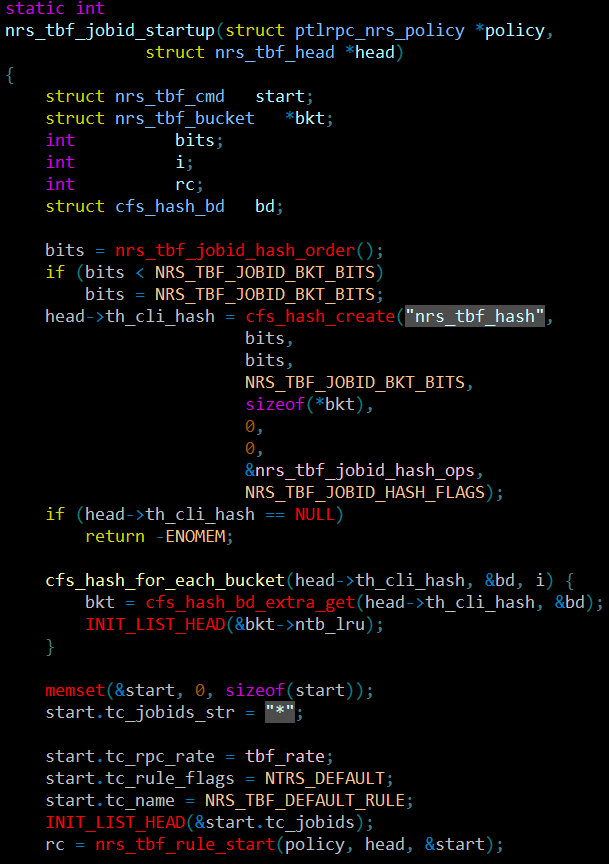
**ptlrpc\_nrs\_req\_add流程分析**



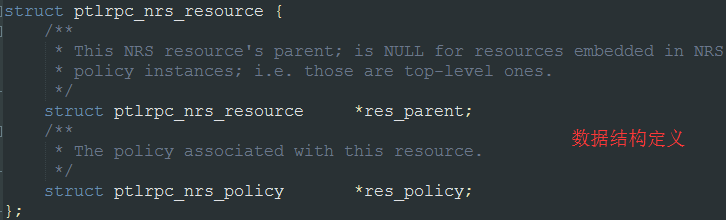
上面的nrs\_request\_enqueue函数调用某种policy的pd\_ops(见下面截图),例如nrs\_tbf\_req\_add，该函数在nrs\_xxx\_ops中绑定，在这里xxx是tbf



下面以tbf为例：那么实际上调用的是nrs\_tbf\_req\_add，该函数会调用cfs\_binheap\_insert，该函数进而调用cfs\_binheap\_bubble，在cfs\_binheap\_bubble中调用hop\_compare,这个函数实际上就是tbf\_cli\_compare,它决定了请求的进入顺序

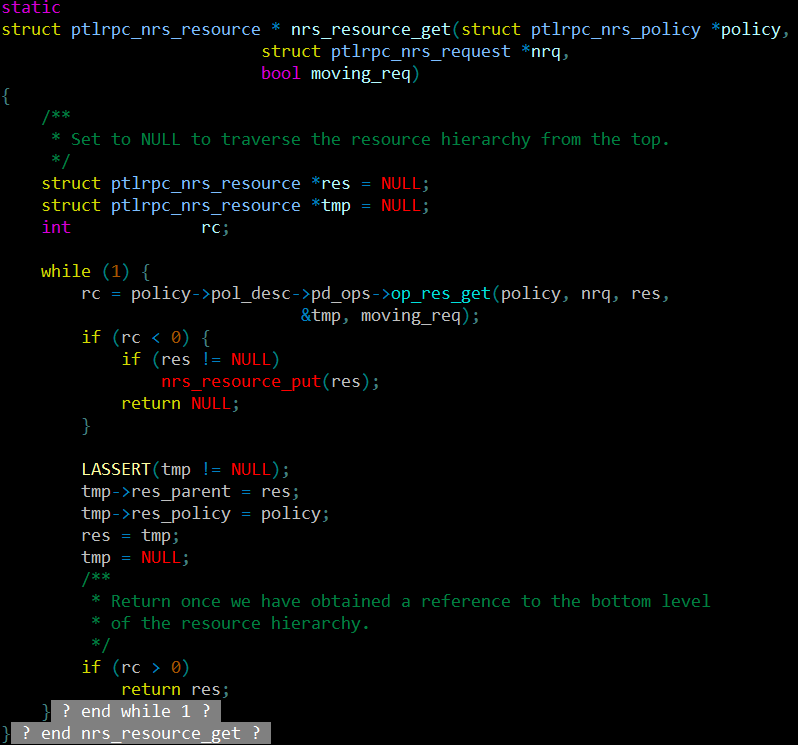


**nrs\_tbf\_head的产生、初始化以及赋给policy->pol\_private**



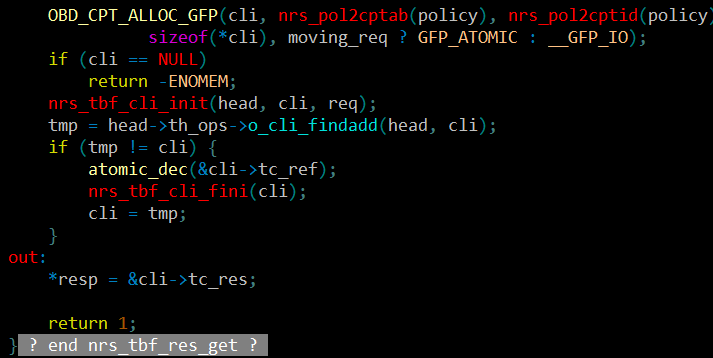
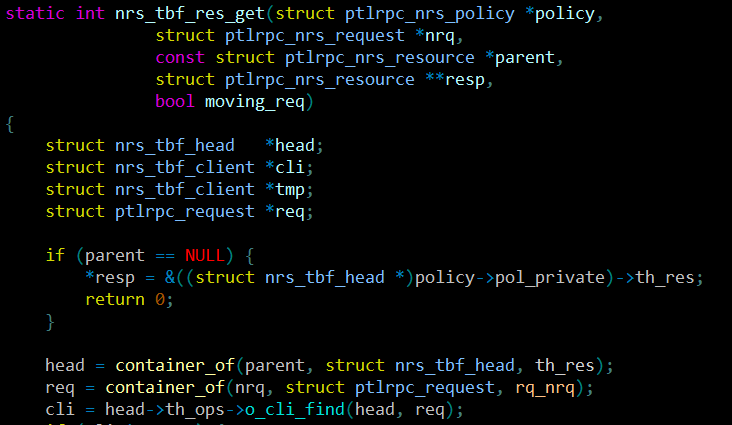
ptlrpc\_nrs\_resource初始化其实这涉及到对其中的两个成员变量的赋值，在nrs\_resource\_get函数中，最开始res和tmp都是空，再看nrs\_tbf\_res\_get代码中，当parent为NULL时，直接返回的是对应nrs\_tbf\_head中th\_res的指针，这里返回这个指针并不是要使用res，而是要给nrs\_tbf\_head中th\_res赋值，相当于是对th\_res初始化（对nrs\_tbf\_head初始化时，其成员变量th\_res并没有初始化）。返回th\_res的地址后，就将th\_res的parent设置为空，它的policy设置为给出的policy。这个while(1)循环其实也只会进行两次，因为下一次循环中nrs\_tbf\_res\_get就会返回1，循环就终止了。最终形成一个两级的ptlrpc\_nrs\_resource结构

**ptlrpc\_nrs\_resource**



因为这句，最多执行不过2次

追根溯源 这里的policy最终是注册的时候赋予的



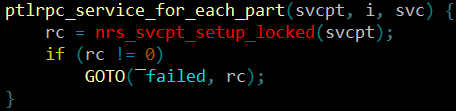
其它代码

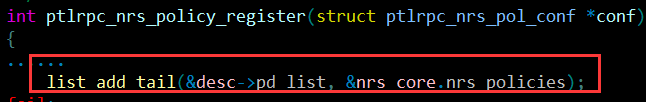
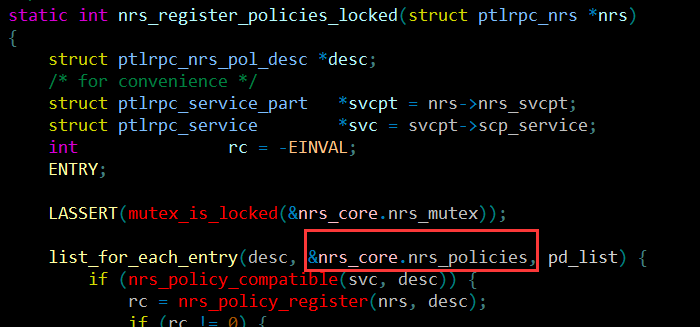
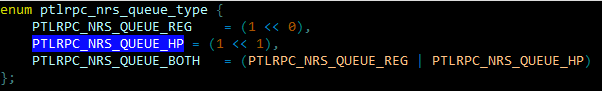
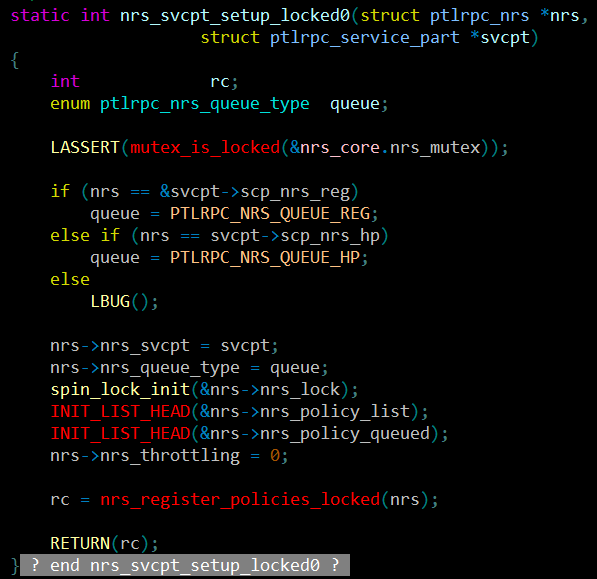
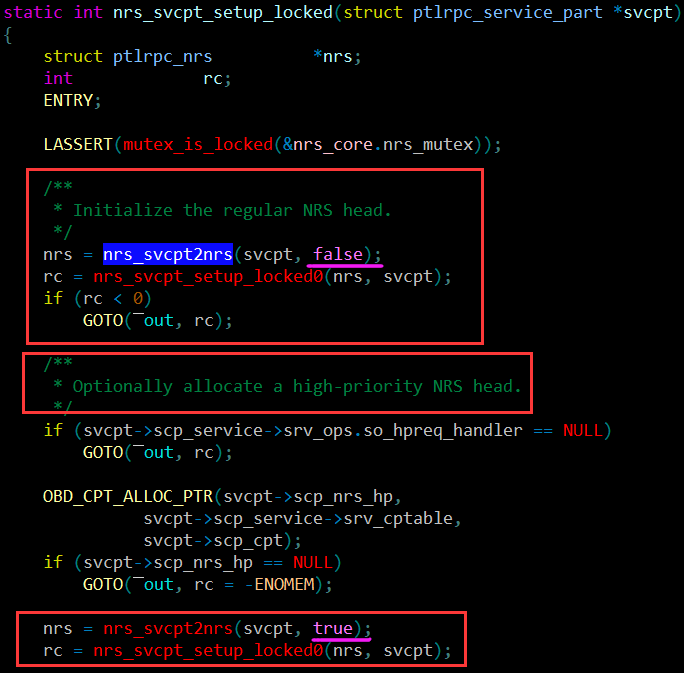


ptlrpc\_service\_nrs\_setup：

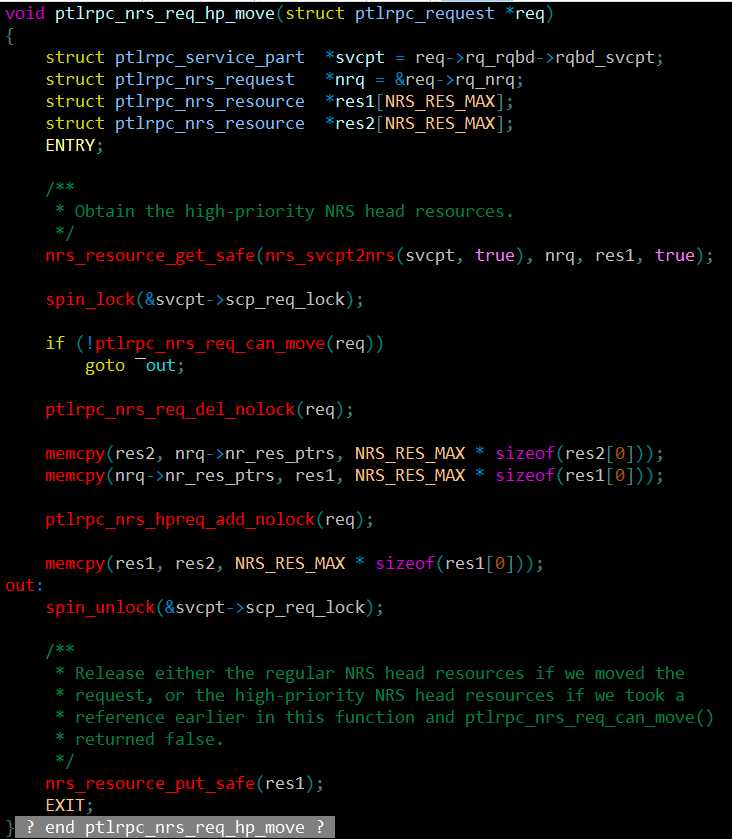
Setup NRS heads on all service partitions of service a svc, and register

all compatible policies on those NRS heads.





追溯到注册



1. 申请较高优先级ptrlrpc\_nrs\_request，并存于res1；
2. 当前req的res保存到res2；
3. 将刚刚申请的保存在res1的高优先级的res复制到req的res中去，这时候req的res就已经被替换的，转变为高优先级的res；
4. 将res2也就是先前req的res，存到res1中
5. 释放res1