Cache Backend API

The FS-Cache system provides an API by which actual caches can be supplied to FS-Cache for it to then serve out to network filesystems and other interested parties. This API is used by:

```
#include <linux/fscache-cache.h>.
```

Overview

Interaction with the API is handled on three levels: cache, volume and data storage, and each level has its own type of cookie object:

COOKIE	C TYPE
Cache cookie	struct fscache_cache
Volume cookie	struct fscache_volume
Data storage cookie	struct fscache_cookie

Cookies are used to provide some filesystem data to the cache, manage state and pin the cache during access in addition to acting as reference points for the API functions. Each cookie has a debugging ID that is included in trace points to make it easier to correlate traces. Note, though, that debugging IDs are simply allocated from incrementing counters and will eventually wrap.

The cache backend and the network filesystem can both ask for cache cookies - and if they ask for one of the same name, they'll get the same cookie. Volume and data cookies, however, are created at the behest of the filesystem only.

Cache Cookies

Caches are represented in the API by cache cookies. These are objects of type:

```
struct fscache_cache {
        void            *cache_priv;
        unsigned int    debug_id;
        char            *name;
        ...
};
```

There are a few fields that the cache backend might be interested in. The `debug_id` can be used in tracing to match lines referring to the same cache and `name` is the name the cache was registered with. The `cache_priv` member is private data provided by the cache when it is brought online. The other fields are for internal use.

Registering a Cache

When a cache backend wants to bring a cache online, it should first register the cache name and that will get it a cache cookie. This is done with:

```
struct fscache_cache *fscache_acquire_cache(const char *name);
```

This will look up and potentially create a cache cookie. The cache cookie may have already been created by a network filesystem looking for it, in which case that cache cookie will be used. If the cache cookie is not in use by another cache, it will be moved into the preparing state, otherwise it will return busy.

If successful, the cache backend can then start setting up the cache. In the event that the initialisation fails, the cache backend should call:

```
void fscache_relinquish_cookie(struct fscache_cache *cache);
```

to reset and discard the cookie.

Bringing a Cache Online

Once the cache is set up, it can be brought online by calling:

```
int fscache_add_cache(struct fscache_cache *cache,
                      const struct fscache_cache_ops *ops,
                      void *cache_priv);
```

This stores the cache operations table pointer and cache private data into the cache cookie and moves the cache to the active state, thereby allowing accesses to take place.

Withdrawing a Cache From Service

The cache backend can withdraw a cache from service by calling this function:

```
void fscache_withdraw_cache(struct fscache_cache *cache);
```

This moves the cache to the withdrawn state to prevent new cache- and volume-level accesses from starting and then waits for outstanding cache-level accesses to complete.

The cache must then go through the data storage objects it has and tell fscache to withdraw them, calling:

```
void fscache_withdraw_cookie(struct fscache_cookie *cookie);
```

on the cookie that each object belongs to. This schedules the specified cookie for withdrawal. This gets offloaded to a workqueue. The cache backend can test for completion by calling:

```
bool fscache_are_objects_withdrawn(struct fscache_cookie *cache);
```

Once all the cookies are withdrawn, a cache backend can withdraw all the volumes, calling:

```
void fscache_withdraw_volume(struct fscache_volume *volume);
```

to tell fscache that a volume has been withdrawn. This waits for all outstanding accesses on the volume to complete before returning.

When the the cache is completely withdrawn, fscache should be notified by calling:

```
void fscache_cache_relinquish(struct fscache_cache *cache);
```

to clear fields in the cookie and discard the caller's ref on it.

Volume Cookies

Within a cache, the data storage objects are organised into logical volumes. These are represented in the API as objects of type:

```
struct fscache_volume {
        struct fscache_cache            *cache;
        void                            *cache_priv;
        unsigned int                    debug_id;
        char                            *key;
        unsigned int                    key_hash;
        ...
        u8                              coherency_len;
        u8                              coherency[];
};
```

There are a number of fields here that are of interest to the caching backend:

- `cache` - The parent cache cookie.
- `cache_priv` - A place for the cache to stash private data.
- `debug_id` - A debugging ID for logging in tracepoints.
- `key` - A printable string with no '/' characters in it that represents the index key for the volume. The key is NUL-terminated and padded out to a multiple of 4 bytes.
- `key_hash` - A hash of the index key. This should work out the same, no matter the cpu arch and endianness.
- `coherency` - A piece of coherency data that should be checked when the volume is bound to in the cache.
- `coherency_len` - The amount of data in the coherency buffer.

Data Storage Cookies

A volume is a logical group of data storage objects, each of which is represented to the network filesystem by a cookie. Cookies are represented in the API as objects of type:

```
struct fscache_cookie {
        struct fscache_volume           *volume;
        void                            *cache_priv;
        unsigned long                   flags;
        unsigned int                    debug_id;
        unsigned int                    inval_counter;
        loff_t                          object_size;
        u8                              advice;
        u32                             key_hash;
        u8                              key_len;
        u8                              aux_len;
        ...
};
```

The fields in the cookie that are of interest to the cache backend are:

- `volume` - The parent volume cookie.

- `cache_priv` - A place for the cache to stash private data.
- `flags` - A collection of bit flags, including:
    - FSCACHE_COOKIE_NO_DATA_TO_READ - There is no data available in the cache to be read as the cookie has been created or invalidated.
    - FSCACHE_COOKIE_NEEDS_UPDATE - The coherency data and/or object size has been changed and needs committing.
    - FSCACHE_COOKIE_LOCAL_WRITE - The netfs's data has been modified locally, so the cache object may be in an incoherent state with respect to the server.
    - FSCACHE_COOKIE_HAVE_DATA - The backend should set this if it successfully stores data into the cache.
    - FSCACHE_COOKIE_RETIRED - The cookie was invalidated when it was relinquished and the cached data should be discarded.
- `debug_id` - A debugging ID for logging in tracepoints.
- `inval_counter` - The number of invalidations done on the cookie.
- `advice` - Information about how the cookie is to be used.
- `key_hash` - A hash of the index key. This should work out the same, no matter the cpu arch and endianness.
- `key_len` - The length of the index key.
- `aux_len` - The length of the coherency data buffer.

Each cookie has an index key, which may be stored inline to the cookie or elsewhere. A pointer to this can be obtained by calling:

```
void *fscache_get_key(struct fscache_cookie *cookie);
```

The index key is a binary blob, the storage for which is padded out to a multiple of 4 bytes.

Each cookie also has a buffer for coherency data. This may also be inline or detached from the cookie and a pointer is obtained by calling:

```
void *fscache_get_aux(struct fscache_cookie *cookie);
```

Cookie Accounting

Data storage cookies are counted and this is used to block cache withdrawal completion until all objects have been destroyed. The following functions are provided to the cache to deal with that:

```
void fscache_count_object(struct fscache_cache *cache);
void fscache_uncount_object(struct fscache_cache *cache);
void fscache_wait_for_objects(struct fscache_cache *cache);
```

The count function records the allocation of an object in a cache and the uncount function records its destruction. Warning: by the time the uncount function returns, the cache may have been destroyed.

The wait function can be used during the withdrawal procedure to wait for fscache to finish withdrawing all the objects in the cache. When it completes, there will be no remaining objects referring to the cache object or any volume objects.

Cache Management API

The cache backend implements the cache management API by providing a table of operations that fscache can use to manage various aspects of the cache. These are held in a structure of type:

```
struct fscache_cache_ops {
        const char *name;
        ...
};
```

This contains a printable name for the cache backend driver plus a number of pointers to methods to allow fscache to request management of the cache:

- Set up a volume cookie [optional]:

    ```
    void (*acquire_volume)(struct fscache_volume *volume);
    ```

    This method is called when a volume cookie is being created. The caller holds a cache-level access pin to prevent the cache from going away for the duration. This method should set up the resources to access a volume in the cache and should not return until it has done so.

    If successful, it can set `cache_priv` to its own data.

- Clean up volume cookie [optional]:

```
void (*free_volume)(struct fscache_volume *volume);
```

This method is called when a volume cookie is being released if `cache_priv` is set.

- Look up a cookie in the cache [mandatory]:

```
bool (*lookup_cookie)(struct fscache_cookie *cookie);
```

This method is called to look up/create the resources needed to access the data storage for a cookie. It is called from a worker thread with a volume-level access pin in the cache to prevent it from being withdrawn.

True should be returned if successful and false otherwise. If false is returned, the withdraw_cookie op (see below) will be called.

If lookup fails, but the object could still be created (e.g. it hasn't been cached before), then:

```
void fscache_cookie_lookup_negative(
        struct fscache_cookie *cookie);
```

can be called to let the network filesystem proceed and start downloading stuff whilst the cache backend gets on with the job of creating things.

If successful, `cookie->cache_priv` can be set.

- Withdraw an object without any cookie access counts held [mandatory]:

```
void (*withdraw_cookie)(struct fscache_cookie *cookie);
```

This method is called to withdraw a cookie from service. It will be called when the cookie is relinquished by the netfs, withdrawn or culled by the cache backend or closed after a period of non-use by fscache.

The caller doesn't hold any access pins, but it is called from a non-reentrant work item to manage races between the various ways withdrawal can occur.

The cookie will have the `FSCACHE_COOKIE_RETIRED` flag set on it if the associated data is to be removed from the cache.

- Change the size of a data storage object [mandatory]:

```
void (*resize_cookie)(struct netfs_cache_resources *cres,
                      loff_t new_size);
```

This method is called to inform the cache backend of a change in size of the netfs file due to local truncation. The cache backend should make all of the changes it needs to make before returning as this is done under the netfs inode mutex.

The caller holds a cookie-level access pin to prevent a race with withdrawal and the netfs must have the cookie marked in-use to prevent garbage collection or culling from removing any resources.

- Invalidate a data storage object [mandatory]:

```
bool (*invalidate_cookie)(struct fscache_cookie *cookie);
```

This is called when the network filesystem detects a third-party modification or when an O_DIRECT write is made locally. This requests that the cache backend should throw away all the data in the cache for this object and start afresh. It should return true if successful and false otherwise.

On entry, new I O/operations are blocked. Once the cache is in a position to accept I/O again, the backend should release the block by calling:

```
void fscache_resume_after_invalidation(struct fscache_cookie *cookie);
```

If the method returns false, caching will be withdrawn for this cookie.

- Prepare to make local modifications to the cache [mandatory]:

```
void (*prepare_to_write)(struct fscache_cookie *cookie);
```

This method is called when the network filesystem finds that it is going to need to modify the contents of the cache due to local writes or truncations. This gives the cache a chance to note that a cache object may be incoherent with respect to the server and may need writing back later. This may also cause the cached data to be scrapped on later rebinding if not properly committed.

- Begin an operation for the netfs lib [mandatory]:

```
bool (*begin_operation)(struct netfs_cache_resources *cres,
                        enum fscache_want_state want_state);
```

This method is called when an I/O operation is being set up (read, write or resize). The caller holds an access pin on the cookie and must have marked the cookie as in-use.

If it can, the backend should attach any resources it needs to keep around to the netfs_cache_resources object and

return true.

If it can't complete the setup, it should return false.

The want_state parameter indicates the state the caller needs the cache object to be in and what it wants to do during the operation:

- `FSCACHE_WANT_PARAMS` - The caller just wants to access cache object parameters; it doesn't need to do data I/O yet.
- `FSCACHE_WANT_READ` - The caller wants to read data.
- `FSCACHE_WANT_WRITE` - The caller wants to write to or resize the cache object.

Note that there won't necessarily be anything attached to the cookie's cache_priv yet if the cookie is still being created.

Data I/O API

A cache backend provides a data I/O API by through the netfs library's `struct netfs_cache_ops` attached to a `struct netfs_cache_resources` by the `begin_operation` method described above.

See the Documentation/filesystems/netfs_library.rst for a description.

Miscellaneous Functions

FS-Cache provides some utilities that a cache backend may make use of:

- Note occurrence of an I/O error in a cache:

  ```
  void fscache_io_error(struct fscache_cache *cache);
  ```

  This tells FS-Cache that an I/O error occurred in the cache. This prevents any new I/O from being started on the cache.

  This does not actually withdraw the cache. That must be done separately.

- Note cessation of caching on a cookie due to failure:

  ```
  void fscache_caching_failed(struct fscache_cookie *cookie);
  ```

  This notes that a the caching that was being done on a cookie failed in some way, for instance the backing storage failed to be created or invalidation failed and that no further I/O operations should take place on it until the cache is reset.

- Count I/O requests:

  ```
  void fscache_count_read(void);
  void fscache_count_write(void);
  ```

  These record reads and writes from/to the cache. The numbers are displayed in /proc/fs/fscache/stats.

- Count out-of-space errors:

  ```
  void fscache_count_no_write_space(void);
  void fscache_count_no_create_space(void);
  ```

  These record ENOSPC errors in the cache, divided into failures of data writes and failures of filesystem object creations (e.g. mkdir).

- Count objects culled:

  ```
  void fscache_count_culled(void);
  ```

  This records the culling of an object.

- Get the cookie from a set of cache resources:

  ```
  struct fscache_cookie *fscache_cres_cookie(struct netfs_cache_resources *cres)
  ```

  Pull a pointer to the cookie from the cache resources. This may return a NULL cookie if no cookie was set.

API Function Reference

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\caching\(linux-master)(Documentation)(filesystems)(caching)backend-api.rst, line 479)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/fscache-cache.h
```