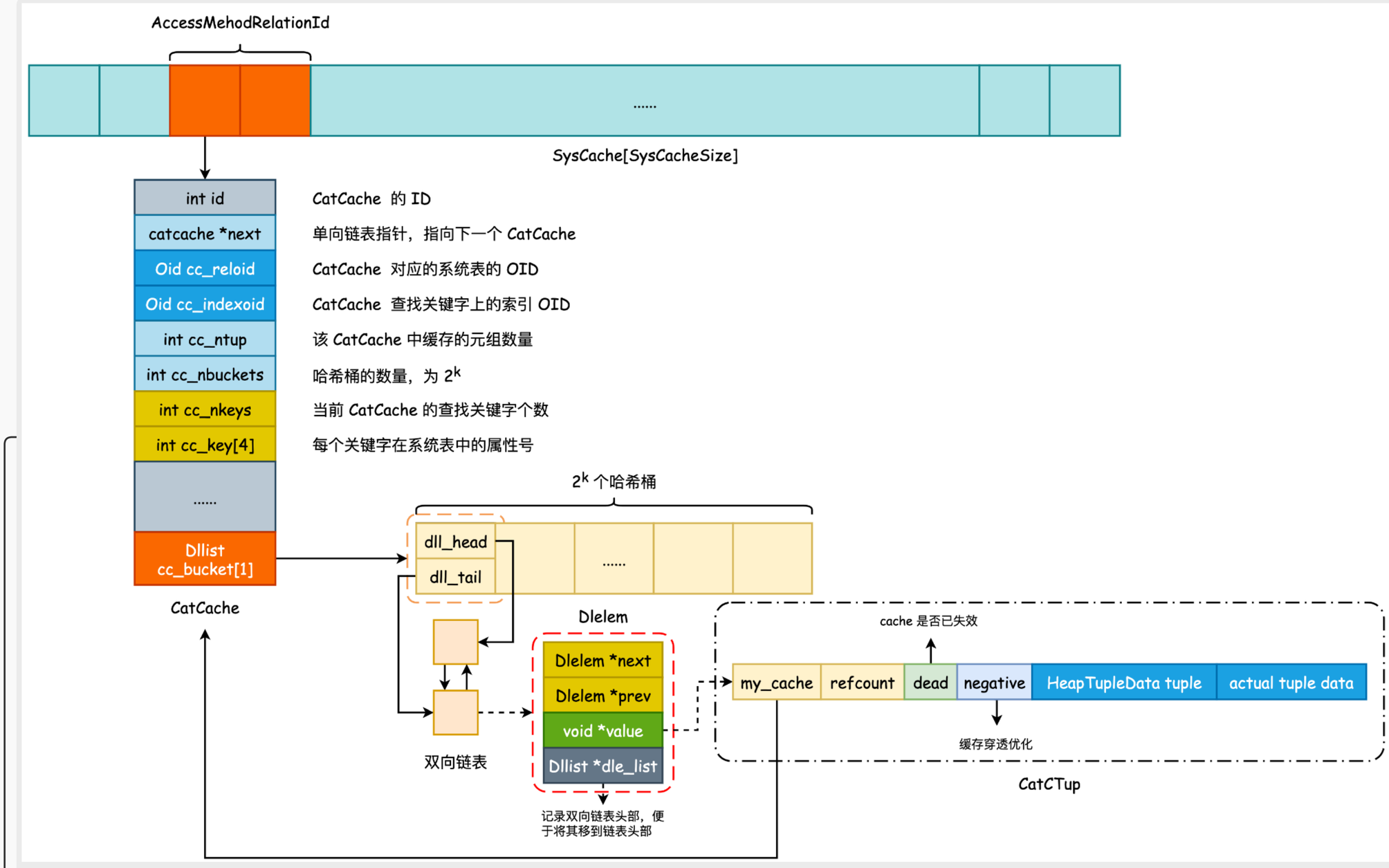
SysCache

作用

- SysCache 用于缓存系统表中的一个一个的 Tuple，例如 pg_class、pg_type、pg_attribute 等经常访问的关系"元数据"，优化数据库对元数据的访问效率
- 设计背景
  - 多表查询
    - 🤔 PostgreSQL 中存在许多张系统表，每一张表拥有不同数量的 tuple，并且行结构也不尽相同。那么为了精简接口，我们希望 SysCache 能够对系统表之间的不同进行抽象，抹去它们之间的差异，并提供统一的接口
  - 多维度查询
    - 🤔 SysCache 不同于其它简单 Key-Value 缓存组件，它需要支持多种维度的查询
    - 例如当我们在 pg_type 这张表查询一个类型信息时，我们希望除了使用 OID 进行查找以外，还能支持类型名称(name)的查找。就像下面两条 SQL 所做的事情一样

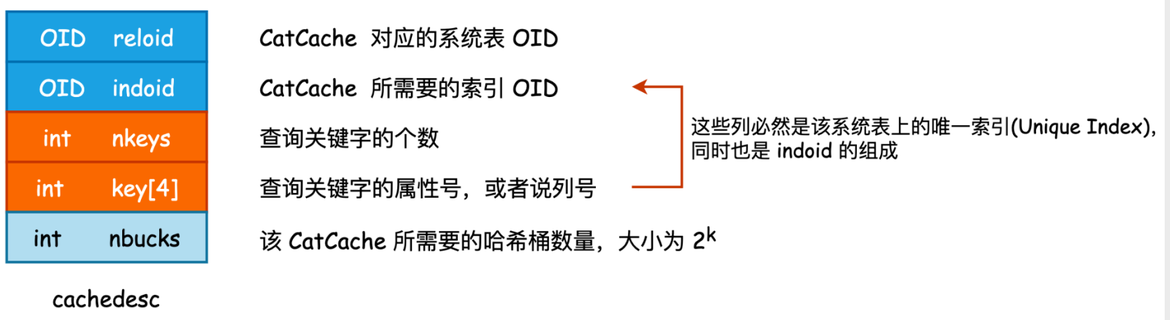
```
select * from pg_type where oid = 25;
select * from pg_type where typname = 'text';
```

- **同时，既然这是一个 K-V 缓存系统，那么必然会使用 HashTable 实现，那么也就有了并发访问的问题，从而牵扯出并发安全、锁、分段锁等相当复杂的问题。PostgreSQL 直接釜底抽薪，将 SysCache 设计成会话级别，多个子进程之间不共享 SysCache**
  - 没有并发，也就不需要锁



实现

- **SysCache 是一个全局静态数组，其大小为 SysCacheSize，由定义在 syscache.c 中的 cacheinfo 静态数组决定。其大小要超过系统表的数量，这是因为需要满足多维度的查询，那么一个系统表在 SysCache 中就会存在多个元素**
- SysCache
  - cacheinfo
    - cacheinfo 是一个静态数组，用于预定义系统需要支持的系统表、查找键以及查找键上的相关索引



    - 这里解释一下 indoid、nkeys 和 key 这三者之间的关系
      - nkeys 和 key 数组组成查询 key，最多支持 4 个查询列的组合
      - **一个系统表有多少个查询维度，在 SysCache 中就有多少个元素**
      - indoid 则是唯一索引，是系统表给 SysCache 的一个承诺：使用这些 key 至多找到一条数据

```
{TypeRelationId,            /* TYPENAMENSP */
    TypeNameNspIndexId,
    2,
    {
        Anum_pg_type_typname,
        Anum_pg_type_typnamespace,
        0,
        0
    },
    1024
},
```

```
DECLARE_UNIQUE_INDEX(pg_type_typname_nsp_index, 2704, on pg_type using btree(typname name_ops, typnamespace oid_ops));
#define TypeNameNspIndexId  2704
```

    - 多维度查询
      - 对于不同维度的查询，我们在调用 API 时将会指定不同的 Search Key
      - **以 TypeRelationId 为例，可以支持通过 Type OID 进行查询，也支持使用 Type Name 进行查询，那么在查询时，我们只需要指定对应 Search Key 在 cacheinfo 中的索引即可，而这个索引，正是由 SysCacheIdentifier 这个枚举类维护的**
      - SearchSysCache1(TYPEOID, oid)
      - SearchSysCache2(TYPENAMENSP, name, namespace)
      - TYPENAMENSP 和 TYPEOID 实际上就是 cacheinfo 数组中的索引，所以，硬编码查询即可
    - 在数据库启动时，只会初始化 SysCache 的内存空间，并无实际的 Tuple 填充，而是随着系统的运行而逐渐增多
  - CatCache
    - SysCache 数组中的成员，保存了某个系统表以某一个维度的缓存数据。例如对于 pg_type 系统表而言，SysCache 中就存在两个元素，一个使用 OID 进行查询，另一个则使用类型名称(name)进行查询
      - 查询维度信息体现在 cc_nkeys 以及 cc_key 这两个主要字段上
    - **cc_bucket 是一个可变长度的数组，其大小必须为 2 的 k 次幂，这更有利于哈希表的优化。PostgreSQL 同样使用拉链法来解决哈希冲突，并且运用 LRU 的一部分功能，将最近使用过的元素移动到双向链表的头部，以便下次更快地返回**
  - Dlelem
    - Dlelem 为拉链法中双向链表元素，记录的 prev 和 next 指针，以及真正的 SysCache Tuple 指针。该字段还额外记录了双向链表的头指针，就是为了以 O(1) 的时间复杂度将该元素移动至链表的头部
  - CatCTup
    - 最终的 Tuple 数据，包括元组的 Header + Data
    - 当缓存数据被删除时，并不会直接从哈希表中移除，而是等到无人再使用它，也就是引用计数变为 0 时，才进行移除
    - 同时，为了优化缓存穿透的问题，即避免因为查询数据本身就不存在，而需要反复到物理表中进行查找的开销，PostgreSQL 对其添加了 negative 字段。当其为真时，表示该数据不存在，缓存+物理表均不存在，直接返回 NULL 即可

API 的使用

- 缓存的精确匹配由函数 SearchCatCache() 所实现，其函数原型为
  - HeapTuple SearchCatCache(CatCache *cache, Datum v1, Datum v2, Datum v3, Datum v4)
  - 其中 v1、v2、v3、v4 用于查找元组的 Key，和 CacheDesc 中的 nkeys 是一一对应的
- 那么现在我们就应该明白 SearchSysCacheX() 这一系列宏定义的使用了
  - SearchSysCache1() 表示只使用 1 个 key 来查找 tuple，SearchSysCache2() 则表示使用 2 个 key 来查找 tuple