前言

缓存，在我们日常开发中是必不可少的一种解决性能问题的方法。简单的说，cache 就是为了提升系统性能而开辟的一块内存空间。

缓存的主要作用是暂时在内存中保存业务系统的数据处理结果，并且等待下次访问使用。在日常开发的很多场合，由于受限于硬盘IO的性能或者我们自身业务系统的数据处理和获取可能非常费时，当我们发现我们的系统这个数据请求量很大的时候，频繁的IO和频繁的逻辑处理会导致硬盘和CPU资源的瓶颈出现。缓存的作用就是将这些来自不易的数据保存在内存中，当有其他线程或者客户端需要查询相同的数据资源时，直接从缓存的内存块中返回数据，这样不但可以提高系统的响应时间，同时也可以节省对这些数据的处理流程的资源消耗，整体上来说，系统性能会有大大的提升。

缓存在很多系统和架构中都用广泛的应用,例如：

1.CPU缓存

2.操作系统缓存

3.本地缓存

4.分布式缓存

5.HTTP缓存

6.数据库缓存

等等，可以说在计算机和网络领域，缓存无处不在。可以这么说，只要有硬件性能不对等，涉及到网络传输的地方都会有缓存的身影。

Guava Cache是一个全内存的本地缓存实现，它提供了线程安全的实现机制。

Guava Cache有两种创建方式：

1. cacheLoader
2. callable callback

LoadingCache是附带CacheLoader构建而成的缓存实现。创建自己的CacheLoader通常只需要简单地实现V load(K key) throws Exception方法

```
1. public class Cache {
2.
3.     private static Map<String, Object> cacheMap =
Maps.newHashMap();
4.
5.     public static Object getValue(String key) {
6.         System.out.println("get From Other");
7.         Integer i = new Random().nextInt();
```

```
8.          cacheMap.put("key", i);
9.          System.out.println("put end " + i);
10.          return cacheMap.get(key);
11.      }
12.
13.      public static void main(String args[]) {
14.
15.          LoadingCache<String, Object> cahceBuilder =
CacheBuilder.newBuilder().refreshAfterWrite(10, TimeUnit.SECONDS)
16.                  .build(new CacheLoader<String, Object>() {
17.                      @Override
18.                      public Object load(String key) throws
Exception {
19.                          return getValue(key);
20.                      }
21.                  });
22.
23.          new Thread(() -> {
24.              try {
25.                  while (true) {
26.                      System.out.println("get : " +
cahceBuilder.get("key"));
27.                      Thread.sleep(2000);
28.                  }
29.              } catch (InterruptedException e) {
30.                  e.printStackTrace();
31.              } catch (ExecutionException e) {
32.                  e.printStackTrace();
33.              }
34.          }).start();
35.      }
36. }
```

从cahceBuilder 获取数据的时候，会判断该key是否有值，如果没有就会调用load方法去获取值，并且存入缓存。
还支持定时刷新，过期等

callable callback的方式较为灵活，允许你在get的时候指定。

```
1. public void testcallableCache()throws Exception{
2.          Cache<String, String> cache =
CacheBuilder.newBuilder().maximumSize(1000).build();
3.          String resultVal = cache.get("jerry", new
Callable<String>() {
4.              public String call() {
5.                  String strProValue="hello "+"jerry"+"!";
6.                  return strProValue;
7.              }
8.          });
9.          System.out.println("jerry value : " + resultVal);
10.
11.          resultVal = cache.get("peida", new Callable<String>()
{
12.              public String call() {
13.                  String strProValue="hello "+"peida"+"!";
```

```
                return strProValue;
            }
        });
        System.out.println("peida value : " + resultVal);
    }

    输出:
    jerry value : hello jerry!
    peida value : hello peida!
```

缓存回收

一个残酷的现实是，我们几乎一定没有足够的内存缓存所有数据。你你必须决定：什么时候某个缓存项就不值得保留了？Guava Cache提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。

基于容量的回收（size-based eviction）

只需使用CacheBuilder.maximumSize(long)。缓存将尝试回收最近没有使用或总体上很少使用的缓存项。——*警告*：在缓存项的数目达到限定值之前，缓存就可能进行回收操作——通常来说，这种情况发生在缓存项的数目逼近限定值时。

定时回收（Timed Eviction）

CacheBuilder提供两种定时回收的方法：

- expireAfterAccess(long, TimeUnit)：缓存项在给定时间内没有被读/写访问，则回收。请注意这种缓存的回收顺序和基于大小回收一样。
- expireAfterWrite(long, TimeUnit)：缓存项在给定时间内没有被写访问（创建或覆盖），则回收。如果认为缓存数据总是在固定时候后变得陈旧不可用，这种回收方式是可取的。

基于引用的回收（Reference-based Eviction）

通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache可以把缓存设置为允许垃圾回收：

- CacheBuilder.weakKeys()：使用弱引用存储键。当键没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用键的缓存用==而不是equals比较键。

- CacheBuilder.weakValues()：使用弱引用存储值。当值没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用值的缓存用==而不是equals比较值。
- CacheBuilder.softValues()：使用软引用存储值。软引用只有在响应内存需要时，才按照全局最近最少使用的顺序回收。考虑到使用软引用的性能影响，我们通常建议使用更有性能预测性的缓存大小限定（见上文，基于容量回收）。使用软引用值的缓存同样用==而不是equals比较值。

其他特性

统计

CacheBuilder.recordStats()用来开启Guava Cache的统计功能。统计打开后，Cache.stats()方法会返回CacheStats对象以提供如下统计信息：

- hitRate()：缓存命中率；
- averageLoadPenalty()：加载新值的平均时间，单位为纳秒；
- evictionCount()：缓存项被回收的总数，不包括显式清除。

此外，还有其他很多统计信息。这些统计信息对于调整缓存设置是至关重要的，在性能要求高的应用中我们建议密切关注这些数据。

asMap视图

asMap视图提供了缓存的ConcurrentMap形式，但asMap视图与缓存的交互需要注意：

- cache.asMap()包含当前所有加载到缓存的项。因此相应地，cache.asMap().keySet()包含当前所有已加载键；
- asMap().get(key)实质上等同于cache.getIfPresent(key)，而且不会引起缓存项的加载。这和Map的语义约定一致。
- 所有读写操作都会重置相关缓存项的访问时间，包括Cache.asMap().get(Object)方法和Cache.asMap().put(K, V)方法，但不包括

Cache.asMap().containsKey(Object)方法，也不包括在Cache.asMap()的集合视图上的操作。比如，遍历Cache.asMap().entrySet()不会重置缓存项的读取时间。