Java开发利器Guava Cache之使用篇

我有一只喵喵 Lv2
2021年10月02日 21:26 · 阅读 779

关注

小知识，大挑战！本文正在参与“程序员必备小知识”创作活动。

前言

提到缓存，可能第一时间想到的就是Redis、Memcache等，这些都属于是分布式缓存，而在某些场景下我们可能并不需要分布式缓存，毕竟需要多引入维护一个中间件，那么在数据量小，且访问频繁，或者说一些不会变的静态配置数据我们都可以考虑放置到本地缓存中，那么我们平时是怎么做的呢？相信大家在写或者在读有关本地缓存代码时，都会看到如下实现方式：

java 复制代码

```
private static final Map<K,V> LOCAL_CACHE = new ConcurrentHashMap<>();
```

的确这种方式简单有效，但是带来的弊端就是过于简单，功能也就过于缺乏，而且如果使用不大，将带来可怕的内存溢出，比如谈起缓存，那不得不提**缓存淘汰策略**、**缓存过期策略**等，但是不要着急，强大的Guava工具库已经为我们提供了简单有效的Guava Cache。

值得注意的是，请不要被强大的Guava Cache迷惑双眼，如果你的缓存场景用不到这些缓存的特性，那么ConcurrentHashMap或许是你最好的选择

Guava Cache

> 官方地址：github.com/google/guav...

「Guava Cache能力一览」

「入门使用」

» key对应的缓存值计算方式

缓存无非可能就是缓存那些耗时很长的计算（除了CPU型任务，I/O型也算）出来值，只有第一次从缓存中访问指定key时，才会进行真正的计算，那么Guava Cache就提供三种缓存计算方式，你也可以理解为缓存加载方式，它们分别是**CacheLoader、Callable、直接插入**。

CacheLoader

CacheLoader方式，简单点说就是计算方式作用于所有key，也就是说通过CacheLoader方法创建的Cache，不管你访问哪个key，它的计算方式都是同一个，来看示例：

java 复制代码

```
@Test
public void guavaCacheTest001(){
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().maximumSize(2)
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key+"真正计算了！");
                return "cache-"+key;
            }
        });

    System.out.println(loadingCache.getUnchecked("key1"));
    System.out.println(loadingCache.getUnchecked("key1"));

    System.out.println(loadingCache.getUnchecked("key2"));
    System.out.println(loadingCache.getUnchecked("key2"));
}
```

```
key1真正计算了!
cache-key1
cache-key1
key2真正计算了!
cache-key2
cache-key2
```

在这个例子中，我们通过给CacheBuilder的build方法传入一个CacheLoad的匿名类，该CacheLoad的load方法逻辑为当获取某个缓存key时，如果该key缓存中不存在，则将计算其缓存值的计算方式。从输出我们可以看到，只有缓存第一次访问时才真正执行了值的计算行为，并且每个缓存key的计算方式都一样。

Callable

当对CacheLoader有了认识之后，你可能会想：如果我针对不同的缓存key的计算缓存值方式并不一样，那该怎么办啊！，别急，Callable为你保驾护航：

java 复制代码

```java
@Test
public void testCallable() throws ExecutionException {
    Cache<Object, Object> cache = CacheBuilder.newBuilder().build();
    Object cacheKey1 = cache.get("key1", () -> {
        System.out.println("key1真正计算了");
        return "key1计算方式1";
    });
    System.out.println(cacheKey1);

    cacheKey1 = cache.get("key1",()->{
        System.out.println("key1真正计算了");
        return "key1计算方式1";
    });
    System.out.println(cacheKey1);

    Object cacheKey2 = cache.get("key2", () -> {
        System.out.println("key1真正计算了");
        return "key1计算方式2";
    });
    System.out.println(cacheKey2);

    cacheKey2 = cache.get("key2",()->{
        System.out.println("key1真正计算了");
        return "key1计算方式2";
    });
    System.out.println(cacheKey2);
}
```

```
key1计算方式1
key1计算方式1
key1真正计算了
key1计算方式2
key1计算方式2
```

从例子中可以看到，在调用**get**的时候，可以传入一个**Callable**来为此缓存key提供专门的缓存值计算方式。

直接插入

这种方式计算缓存值的逻辑不再由Guava Cache管理，而是调用方可以调用**put(key,value)** 直接将要缓存的值插入。

java 复制代码

```java
@Test
public void testDirectInsert() throws ExecutionException {
    Cache<Object, Object> cache = CacheBuilder.newBuilder().build();
    cache.put("key1","cache-key1");
    System.out.println(cache.get("key1",()->"callable cache-key1"));
}
```

```
输出：
cache-key1
```

» 缓存淘汰机制

有一个残酷的事实就是，往往我们没有那么大的内存去支撑我们的缓存，所以我们必须有效的利用起来我们这昂贵的内存，即针对那些不常用的缓存及时剔除，那么Guava Cache为我们提供了三种缓存剔除机制：**基于大小剔除、基于缓存时间剔除、基于引用剔除。**

基于大小剔除

这里并不是指占用缓存大小，而是指缓存条目的数量，当缓存key的数量达到指定数量时，将按照LRU针对缓存key进行剔除。

java 复制代码

```java
@Test
public void testSizeBasedEviction(){
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().maximumSize(3)
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key+"真正计算了");
                return "cached-" + key;
            }
```

```
    System.out.println("第一次访问");
    loadingCache.getUnchecked("key1");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key3");

    System.out.println("第二次访问");
    loadingCache.getUnchecked("key1");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key3");

    System.out.println("开始剔除");
    loadingCache.getUnchecked("key4");

    System.out.println("第三次访问");
    loadingCache.getUnchecked("key3");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key1");
}
```

```
输出:
第一次访问
key1真正计算了
key2真正计算了
key3真正计算了
第二次访问
开始剔除
key4真正计算了
第三次访问
key1真正计算了
```

在上面这个例子中，设置了最大缓存条目为3，然后依次添加了三个缓存项，并且依次进行了访问，可以看到当第一次访问时，由于缓存中都没值，因此进行了计算，第二次访问时，由于缓存中都有值所以直接从缓存读取，到了开始剔除阶段时，此时尝试获取之前没访问过的key4，而由于最大缓存条目为3，所以此时需要从缓存中剔除掉一个值，那么剔除谁呢？遵循LRU算法，key1是最近最不常不使用的，所以剔除的就是key1了，从我们第三次访问输出的结果就可以验证。

注意：如果maximumSize传入0，则所有key都将不进行缓存！

除了maximumSize指定缓存key最大数量，也可以通过**maximumWeight**指定最大权重，就是说，每个缓存的key都需要返回一个权重，如果所有缓存的key的权重之和大于了我们指定的最大权重，那么将执行LRU淘汰策略：

java 复制代码

```
@Test
public void testWeightBasedEviction(){
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().maximumWeight(6).weigher((key,value
```

```
        }
        if (key.equals("key2")){
            return 2;
        }
        if (key.equals("key3")){
            return 3;
        }

        if (key.equals("key4")){
            return 1;
        }
        return 0;
    })
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key+"真正计算了");
                return "cached-" + key;
            }
        });

    System.out.println("第一次访问");
    loadingCache.getUnchecked("key1");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key3");

    System.out.println("第二次访问");
    loadingCache.getUnchecked("key1");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key3");

    System.out.println("开始剔除");
    loadingCache.getUnchecked("key4");
    loadingCache.getUnchecked("key3");
    loadingCache.getUnchecked("key2");
    loadingCache.getUnchecked("key1");
}
```

输出：

```
第一次访问
key1真正计算了
key2真正计算了
key3真正计算了
第二次访问
开始剔除
key4真正计算了
key1真正计算了
```

这个就不多解释了吧，自己根据输出想想...

基于时间剔除

Guava Cache针对CacheBuilder提供了两个方法：**expireAfterAccess(long, TimeUnit)** 和 **expireAfterWrite(long, TimeUnit)**

- expireAfterAccess

顾名思义，当某个缓存key自最后一次访问（读取或者写入）超过指定时间后，那么这个缓存key将失效。

java 复制代码

```
@Test
public void testExpiredAfterAccess() throws InterruptedException {
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().expireAfterAccess(3,TimeUnit.SECOND
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key+"真正计算了");
                return "cached-" + key;
            }
        });

    System.out.println("第一次访问（写入）");
    loadingCache.getUnchecked("key1");

    System.out.println("第二次访问");
    loadingCache.getUnchecked("key1");

    TimeUnit.SECONDS.sleep(3);
    System.out.println("过3秒后访问");
    loadingCache.getUnchecked("key1");
}

输出:
第一次访问（写入）
key1真正计算了
第二次访问
过3秒后访问
key1真正计算了
```

这个例子中，我们设置了缓存自最近一次访问（或写入）超过3秒后，将失效，通过输出也可以看到确实 如此。

- expireAfterWrite

顾名思义，当缓存key自最近一次写入（**注意，这就是和expireAfterAccess的区别，expireAfterWrite强调写，不关心读**）超过一定时间则过期剔除：

java 复制代码

```
@Test
public void testExpiredAfterWrite() throws InterruptedException {
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().expireAfterWrite(3,TimeUnit.SECONDS
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key+"真正计算了");
                return "cached-" + key;
            }
        });
    for (int i = 0; i < 4; i++) {
        System.out.println(new Date());
        loadingCache.getUnchecked("key1"); //首次执行的时候，为写入
        TimeUnit.SECONDS.sleep(1);
    }
}
```

输出：

```
Sat Oct 02 20:06:47 CST 2021
key1真正计算了
Sat Oct 02 20:06:48 CST 2021
Sat Oct 02 20:06:49 CST 2021
Sat Oct 02 20:06:50 CST 2021
key1真正计算了
```

同样，这里根据程序和输出应该可以理解啦！

基于引用剔除

Java有四大引用，强、软、弱、虚、如果对这几个引用不是很了解的可以先去看看我这篇文章：Java四种引用类型：强、软、弱、虚

Guava Cache提供了基于引用的剔除策略，看到这里，你是否想起来了**ThreadLocal如何防止内存泄露呢?**，如果不知道没关系，继续看我上面贴的引用文章。Guava Cache提供了三种基于引用剔除的策略：

- CacheBuilder.weakKeys()

当我们使用了**weakKeys()** 后，Guava cache将以**弱引用** 的方式去存储缓存key,那么根据弱引用的定义：**当发生垃圾回收时，不管当前系统资源是否充足，弱引用都会被回收**，直接上例子：

java 复制代码

```
@Test
public void testWeakKeys() throws InterruptedException {
```

```
            @Override
            public String load(MyKey key) throws Exception {
                System.out.println(key.getKey()+"真正计算了");
                return "cached-" + key.getKey();
            }
        });

    MyKey key = new MyKey("key1");
    System.out.println("第一次访问");
    loadingCache.getUnchecked(key);
    System.out.println(loadingCache.asMap());

    System.out.println("第二次访问");
    loadingCache.getUnchecked(key);
    System.out.println(loadingCache.asMap());

    System.out.println("key失去强引用GC后访问");
    key = null;
    System.gc();
    TimeUnit.SECONDS.sleep(3);
    System.out.println(loadingCache.asMap());

}

@Data
private static class MyKey{
    String key;

    public MyKey(String key) {
        this.key = key;
    }
}
```

- CacheBuilder.weakValues()

有了CacheBuilder.weakKeys()的基础，CacheBuilder.weakValues()的作用想必照猫画虎应该也知道了吧？换汤不换药，这次针对的是缓存值！

- CacheBuilder.softValues()

有了CacheBuilder.weakValues()的基础，CacheBuilder.softValues()的作用相比照猫画虎应该也知道了吧？对，你真棒，就是之前的弱引用换为了软引用，软引用相比弱引用，被回收的条件就苛刻点：**当发生垃圾回收时，只有当系统资源不足时，才会回收！**。

主动剔除

- Cache.invalidate(key)
- Cache.invalidateAll(keys)
- Cache.invalidateAll()

缓存失效监听器

有时候我们希望当缓存失效被剔除的时候，可以做一些善后事情，此时，我们就可以通过 **CacheBuilder.removalListener(RemovalListener)** 来指定一个缓存失效监听器，当缓存失效时，将回调我们的监听器：

java 复制代码

```
@Test
public void testRemovalListener() throws InterruptedException {
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().removalListener(notification -> {
        System.out.println(String
            .format("缓存 %s 因为 %s 失效了，它的value是%s", notification.getKey(), notification.getCause(),
                notification.getValue()));
    }).expireAfterAccess(3, TimeUnit.SECONDS).build(new CacheLoader<String, String>() {
        @Override
        public String load(String key) throws Exception {
            System.out.println(key + "真正计算了");
            return "cached-" + key;
        }
    });

    System.out.println("第一次访问（写入）");
    loadingCache.getUnchecked("key1");

    System.out.println("第二次访问");
    loadingCache.getUnchecked("key1");
    TimeUnit.SECONDS.sleep(3);

    System.out.println("3秒后");
    loadingCache.getUnchecked("key1");
}

输出：
第一次访问（写入）
key1真正计算了
第二次访问
3秒后
缓存 key1 因为 EXPIRED 失效了，它的value是cached-key1
key1真正计算了
```

» Guava Cache什么进行清理动作?

这个其实在上节实验缓存剔除监听器的时候我就发现一个问题：如果缓存失效后，我不再进行任何操作，那么这个缓存监听器就得不到调用！，从这里就可以看出，Guava cache并不是自己主动去清理那些失效缓存的，而是当我们对缓存进行了操作时，才会进行检查清理以及其他动作。那么为什么呢？想想啊，如果要主动清除，那肯定要有一个一直运行的后台线程去执行清理，多了个线程出来，那么意味着不再是单线程程序了，涉及多线程就要考虑加锁资源保护了，这无疑会消耗我们资源，影响性能，而主动清除又不是必须的，等你操作了再清除，一点也不晚！

当然Guava cache也提供给我们主动清理的方法：**Cache.cleanUp()**,那么有了这个方法之后，是否主动清理的操作就交给了我们，由我们自己去权衡。

» 缓存刷新

CacheBuilder中提供了**refreshAfterWrite** 用来指定缓存key写入多久后重新进行计算并缓存：

java 复制代码

```
@Test
public void testRefresh() throws InterruptedException {
    LoadingCache<String, String> loadingCache = CacheBuilder.newBuilder().refreshAfterWrite(1,TimeUnit.SECONDS
        .build(new CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                System.out.println(key + "真正计算了");
                return "cached-" + key;
            }
        });
    for (int i = 0; i < 3; i++) {
        loadingCache.getUnchecked("key1");
        TimeUnit.SECONDS.sleep(2);
    }
}

输出
key1真正计算了
key1真正计算了
key1真正计算了
```

在这个例子中，我们指定缓存key写入后，超过1秒就会刷新，然后我们每隔2秒访问一次缓存key,可以看到每次都得到了重新计算！

「小结」

本文通过大量代码案例详细介绍了Guava Cache的使用，当然你以为会止步于此吗？由于篇幅的原因，本文为使用篇，接下来将推出原理篇，我们的目的是从这些大佬的源码设计中吸取精华，所谓知己知彼～

分类： 后端 标签： 后端 Java

评论

输入评论（Enter换行，Ctrl + Enter发送）

相关推荐

_晨曦_ 3年前 Android

Java里的Boost——Google Guava官方教程中文版

1332 12 评论

小饭饭饭饭饭饭 1年前 Java

Spring官方都说废掉GuavaCache用Caffeine，赶快换系列三

1610 6 4

程序员面试之道 11月前 后端

一篇让你熟练掌握Google Guava包(全网最全)

2211 14 评论

潜行前行 10月前 Java 后端

工具篇：介绍几个好用的guava工具类

2025 32 3

郁垒 2年前 Spring Boot

Springboot应用cache，将@Cacheable、@CacheEvict注解应用在mybatis mapper的...

2038 2 6

SpringBoot中文社区 1年前 Spring Boot

在springboot中使用Guava基于令牌桶实现限流

磊仔 3年前 前端 掘金翻译计划

[译] 写给大家看的 Cache-Control 指令配置

3787 43 6

漫话编程 2年前 后端 Java CDN

漫话：如何给女朋友解释什么是CDN?

2.1w 293 26

捡田螺的小男孩 7月前 Java 后端

2W字！详解20道Redis经典面试题！（珍藏版）

2.3w 492 37

暮色妖娆丶 1月前 后端 Java

优秀的后端应该有哪些开发习惯?

3.9w 481 187

码农突围 1年前 Java

为什么强烈推荐 Java 程序员使用 Google Guava 编程

1169 2 评论

CodeSheep 3年前 Spring Boot

Guava Cache本地缓存在 Spring Boot应用中的实践

1180 19 2

harvey_yh 9月前 后端

GuavaCache中解决业务NPE 问题

313 1 评论

我有一只喵喵 6月前 Java 后端

Java开发利器之重试框架guava-retrying

1454 13 评论

谁主沉浮oo7 1年前 Java

使用Guava RateLimiter限流入门到深入

MacroZheng 1月前 后端 Java Redis

颜值爆表！Redis官方可视化工具来啦，功能真心强大！

4.5w 282 41

Life_of_Coder 8月前 后端

Guava Cache实现原理——开篇&基本实现

609 1 评论

小_菜鸟 4年前 服务器 API Java

JAVA RESTful WebService实战笔记(三)

237 点赞 评论

Java技术江湖 2年前 Java

为什么强烈推荐 Java 程序员使用 Google Guava 编程！

826 点赞 评论