如果是想直接看官网教程的请移步：[https://github.com/ben-manes/...](https://link.segmentfault.com/?enc=9ftIR89FyueX1cLqoFZpwA%3D%3D.fdgmjDqXorvRwoLLuNzpjfiJ3DrbwEP%2BRN13KwuObY1ltEkOXBdU4zbIR0dYKBgI" \t "_blank)

**而如果还想结合实际应用场景，以及各种坑的，请看本文。**

最近来了一个实习生小张，看了我在公司项目中使用的缓存框架Caffeine，三天两头跑来找我取经，说是要把Caffeine吃透，为此无奈的也只能一个个细心解答了。

后来这件事情被总监直到了，说是后面还有新人，让我将相关问题和细节汇总成一份教程，权当共享好了，该份教程也算是全网第一份，结合了目前我司游戏中业务场景的应用和思考，以及踩过的坑。

实习生小张：稀饭稀饭，以前我们游戏中应用的缓存其实是谷歌提供的ConcurrentLinkedHashMap，为什么后面你强烈要求换成用Caffeine呢？

关于上面的问题，具体有以下几个原因：

* 使用谷歌提供的ConcurrentLinkedHashMap有个漏洞，那就是缓存的过期只会发生在缓存达到上限的情况，否则便只会一直放在缓存中。咋一看，这个机制没问题，是没问题，可是却不合理，举个例子，有玩家上线后加载了一堆的数据放在缓存中，之后便不再上线了，那么这份缓存便会一直存在，知道缓存达到上限。
* ConcurrentLinkedHashMap没有提供基于时间淘汰时间的机制，而Caffeine有，并且有多种淘汰机制，并且支持淘汰通知。
* 目前Spring也在推荐使用，Caffeine 因使用 Window TinyLfu 回收策略，提供了一个近乎最佳的命中率。

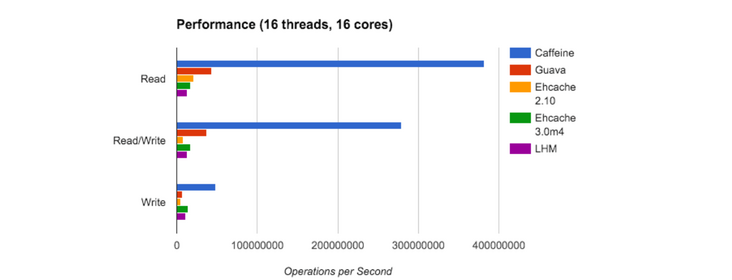
实习生小张：哦哦哦，我了解了，是否可以跟我介绍下Caffeine呢？

可以的，Caffeine是基于Java8的高性能缓存库，可提供接近最佳的命中率。Caffeine的底层使用了ConcurrentHashMap，支持按照一定的规则或者自定义的规则使缓存的数据过期，然后销毁。

再说一个劲爆的消息，很多人都听说过Google的GuavaCache，而没有听说过Caffeine，其实和Caffeine相比，GuavaCache简直就是个弟中弟，这不SpringFramework5.0（SpringBoot2.0）已经放弃了Google的GuavaCache，转而选择了Caffeine。



为什么敢这么夸Caffeine呢？我们可以用官方给出的数据说话。



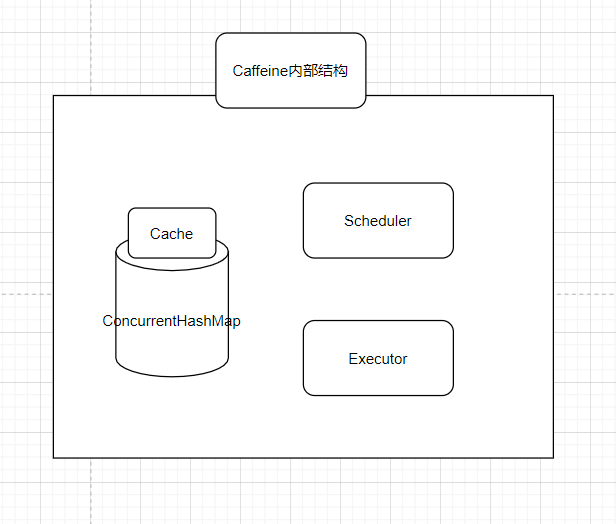
Caffeine提供了多种灵活的构造方法，从而可以创建多种特性的本地缓存。

1. 自动把数据加载到本地缓存中，并且可以配置异步；
2. 基于数量剔除策略；
3. 基于失效时间剔除策略，这个时间是从最后一次操作算起【访问或者写入】；
4. 异步刷新；
5. Key会被包装成Weak引用；
6. Value会被包装成Weak或者Soft引用，从而能被GC掉，而不至于内存泄漏；
7. 数据剔除提醒；
8. 写入广播机制；
9. 缓存访问可以统计；

实习生小张：我擦，这么强大，为什么可以这么强大呢，稀饭你不是自称最熟悉Caffeine的人吗？能否给我大概所说内部结构呢？

我日，我没有，我只是说在我们项目组我最熟悉，别污蔑我

那接下来我大概介绍下Caffeine的内部结构



* Cache的内部包含着一个ConcurrentHashMap，这也是存放我们所有缓存数据的地方，众所周知，ConcurrentHashMap是一个并发安全的容器，这点很重要，可以说Caffeine其实就是一个被强化过的ConcurrentHashMap。
* Scheduler，定期清空数据的一个机制，可以不设置，如果不设置则不会主动的清空过期数据。
* Executor，指定运行异步任务时要使用的线程池。可以不设置，如果不设置则会使用默认的线程池，也就是ForkJoinPool.commonPool()

实习生小张：听起来就是一个强化版的ConcurrentHashMap，那么需要导入什么包吗？

Caffeine的依赖，其实还是很简单的，直接引入maven依赖即可。

<dependency>

<groupId>com.github.ben-manes.caffeine</groupId>

<artifactId>caffeine</artifactId>

</dependency>

实习生小张：可以，导入成功了，你一直和我说Caffeine的数据填充机制设计的很优美，不就是put数据吗？有什么优美的？说说看吗？

是put数据，只是针对put数据，Caffeine提供了三种机制，分别是

* 手动加载
* 同步加载
* 异步加载

我分别举个例子，比如手动加载

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/17 0:16

\* @desc 手动填充数据

\*/

public class CaffeineManualTest {

@Test

public void test() {

// 初始化缓存，设置了1分钟的写过期，100的缓存最大个数

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfterWrite(1, TimeUnit.MINUTES)

.maximumSize(100)

.build();

int key1 = 1;

// 使用getIfPresent方法从缓存中获取值。如果缓存中不存指定的值，则方法将返回 null：

System.out.println(cache.getIfPresent(key1));

// 也可以使用 get 方法获取值，该方法将一个参数为 key 的 Function 作为参数传入。如果缓存中不存在该 key

// 则该函数将用于提供默认值，该值在计算后插入缓存中：

System.out.println(cache.get(key1, new Function<Integer, Integer>() {

@Override

public Integer apply(Integer integer) {

return 2;

}

}));

// 校验key1对应的value是否插入缓存中

System.out.println(cache.getIfPresent(key1));

// 手动put数据填充缓存中

int value1 = 2;

cache.put(key1, value1);

// 使用getIfPresent方法从缓存中获取值。如果缓存中不存指定的值，则方法将返回 null：

System.out.println(cache.getIfPresent(1));

// 移除数据，让数据失效

cache.invalidate(1);

System.out.println(cache.getIfPresent(1));

}

}

上面提到了两个get数据的方式，一个是getIfPercent，没数据会返回Null，而get数据的话则需要提供一个Function对象，当缓存中不存在查询的key则将该函数用于提供默认值，并且会插入缓存中。

实习生小张：如果同时有多个线程进行get，那么这个Function对象是否会被执行多次呢？

实际上不会的，可以从结构图看出，Caffeine内部最主要的数据结构就是一个ConcurrentHashMap，而get的过程最终执行的便是ConcurrentHashMap.compute，这里仅会被执行一次。

接下来说说同步加载数据

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/19 9:47

\* @desc 同步加载数据

\*/

public class CaffeineLoadingTest {

/\*\*

\* 模拟从数据库中读取key

\*

\* @param key

\* @return

\*/

private int getInDB(int key) {

return key + 1;

}

@Test

public void test() {

// 初始化缓存，设置了1分钟的写过期，100的缓存最大个数

LoadingCache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfterWrite(1, TimeUnit.MINUTES)

.maximumSize(100)

.build(new CacheLoader<Integer, Integer>() {

@Nullable

@Override

public Integer load(@NonNull Integer key) {

return getInDB(key);

}

});

int key1 = 1;

// get数据，取不到则从数据库中读取相关数据，该值也会插入缓存中：

Integer value1 = cache.get(key1);

System.out.println(value1);

// 支持直接get一组值，支持批量查找

Map<Integer, Integer> dataMap

= cache.getAll(Arrays.asList(1, 2, 3));

System.out.println(dataMap);

}

}

所谓的同步加载数据指的是，在get不到数据时最终会调用build构造时提供的CacheLoader对象中的load函数，如果返回值则将其插入缓存中，并且返回，这是一种同步的操作，也支持批量查找。

**实际应用：在我司项目中，会利用这个同步机制，也就是在CacheLoader对象中的load函数中，当从Caffeine缓存中取不到数据的时候则从数据库中读取数据，通过这个机制和数据库结合使用**

最后一种便是异步加载

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/19 22:34

\* @desc 异步加载

\*/

public class CaffeineAsynchronousTest {

/\*\*

\* 模拟从数据库中读取key

\*

\* @param key

\* @return

\*/

private int getInDB(int key) {

return key + 1;

}

@Test

public void test() throws ExecutionException, InterruptedException {

// 使用executor设置线程池

AsyncCache<Integer, Integer> asyncCache = Caffeine.newBuilder()

.expireAfterWrite(1, TimeUnit.MINUTES)

.maximumSize(100).executor(Executors.newSingleThreadExecutor()).buildAsync();

Integer key = 1;

// get返回的是CompletableFuture

CompletableFuture<Integer> future = asyncCache.get(key, new Function<Integer, Integer>() {

@Override

public Integer apply(Integer key) {

// 执行所在的线程不在是main，而是ForkJoinPool线程池提供的线程

System.out.println("当前所在线程：" + Thread.currentThread().getName());

int value = getInDB(key);

return value;

}

});

int value = future.get();

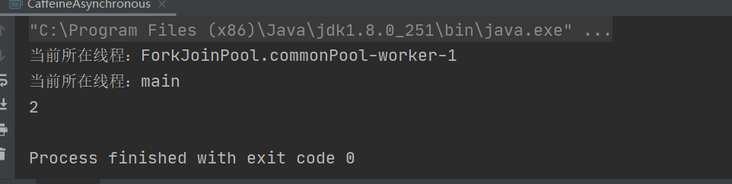
System.out.println("当前所在线程：" + Thread.currentThread().getName());

System.out.println(value);

}

}

执行结果如下



可以看到getInDB是在线程池ForkJoinPool提供的线程中执行的，而且asyncCache.get()返回的是一个CompletableFuture，熟悉流式编程的人对这个会比较熟悉，可以用CompletableFuture来实现异步串行的实现。

实习生小张：我看到默认是线程池ForkJoinPool提供的线程，实际上不大可能用默认的，所以我们可以自己指定吗？

答案是可以的，实例如下

// 使用executor设置线程池

AsyncCache<Integer, Integer> asyncCache = Caffeine.newBuilder()

.expireAfterWrite(1, TimeUnit.MINUTES)

.maximumSize(100).executor(Executors.newSingleThreadExecutor()).buildAsync();

实习生小张：听说Caffeien最屌的地方就是它那健全的淘汰机制，可以说说看吗？

可以的，实际上Caffeine比ConcurrentHashMap和相比，最明显的一点便是提供了一套完整的淘汰机制。

基于基本需求，Caffeine提供了三种淘汰机制：

* 基于大小
* 基于权重
* 基于时间
* 基于引用

基本上这三个对于我们来说已经是够用了，接下来我针对这几个都给出相关的例子。

首先是基于大小淘汰，设置方式：maximumSize(个数)，这意味着当缓存大小超过配置的大小限制时会发生回收。

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/19 22:34

\* @desc 基于大小淘汰

\*/

public class CaffeineSizeEvictionTest {

@Test

public void test() throws InterruptedException {

// 初始化缓存，缓存最大个数为1

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.maximumSize(1)

.build();

cache.put(1, 1);

// 打印缓存个数，结果为1

System.out.println(cache.estimatedSize());

cache.put(2, 2);

// 稍微休眠一秒

Thread.sleep(1000);

// 打印缓存个数，结果为1

System.out.println(cache.estimatedSize());

}

}

我这边设置了最大缓存个数为1，当put进二个数据时则第一个就被淘汰了，此时缓存内个数只剩1个。

之所以在demo中需要休眠一秒，是因为淘汰数据是一个异步的过程，休眠一秒等异步的回收结束。

接下来说说基于权重淘汰的方式，设置方式：maximumWeight(个数)，意味着当缓存大小超过配置的权重限制时会发生回收。

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/21 15:26

\* @desc 基于缓存权重

\*/

public class CaffeineWeightEvictionTest {

@Test

public void test() throws InterruptedException {

// 初始化缓存，设置最大权重为2

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.maximumWeight(2)

.weigher(new Weigher<Integer, Integer>() {

@Override

public @NonNegative int weigh(@NonNull Integer key, @NonNull Integer value) {

return key;

}

})

.build();

cache.put(1, 1);

// 打印缓存个数，结果为1

System.out.println(cache.estimatedSize());

cache.put(2, 2);

// 稍微休眠一秒

Thread.sleep(1000);

// 打印缓存个数，结果为1

System.out.println(cache.estimatedSize());

}

}

我这边设置了最大权重为2，权重的计算方式是直接用key，当put 1 进来时总权重为1，当put 2 进缓存是总权重为3，超过最大权重2，因此会触发淘汰机制，回收后个数只为1。

然后是基于时间的方式，基于时间的回收机制，Caffeine有提供了三种类型，可以分为：

* 访问后到期，时间节点从最近一次读或者写，也就是get或者put开始算起。
* 写入后到期，时间节点从写开始算起，也就是put。
* 自定义策略，自定义具体到期时间。

这三个我举个例子，看好相关的区别了哈。

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/24 23:41

\* @desc 基于时间淘汰

\*/

public class CaffeineTimeEvictionTest {

/\*\*

\* 访问后到期

\*

\* @throws InterruptedException

\*/

@Test

public void testEvictionAfterProcess() throws InterruptedException {

// 设置访问5秒后数据到期

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfterAccess(5, TimeUnit.SECONDS).scheduler(Scheduler.systemScheduler())

.build();

cache.put(1, 2);

System.out.println(cache.getIfPresent(1));

Thread.sleep(6000);

System.out.println(cache.getIfPresent(1));

}

/\*\*

\* 写入后到期

\*

\* @throws InterruptedException

\*/

@Test

public void testEvictionAfterWrite() throws InterruptedException {

// 设置写入5秒后数据到期

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfterWrite(5, TimeUnit.SECONDS).scheduler(Scheduler.systemScheduler())

.build();

cache.put(1, 2);

System.out.println(cache.getIfPresent(1));

Thread.sleep(6000);

System.out.println(cache.getIfPresent(1));

}

/\*\*

\* 自定义过期时间

\*

\* @throws InterruptedException

\*/

@Test

public void testEvictionAfter() throws InterruptedException {

Cache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfter(new Expiry<Integer, Integer>() {

// 创建1秒后过期，可以看到这里必须要用纳秒

@Override

public long expireAfterCreate(@NonNull Integer key, @NonNull Integer value, long currentTime) {

return TimeUnit.SECONDS.toNanos(1);

}

// 更新2秒后过期，可以看到这里必须要用纳秒

@Override

public long expireAfterUpdate(@NonNull Integer key, @NonNull Integer value, long currentTime, @NonNegative long currentDuration) {

return TimeUnit.SECONDS.toNanos(2);

}

// 读3秒后过期，可以看到这里必须要用纳秒

@Override

public long expireAfterRead(@NonNull Integer key, @NonNull Integer value, long currentTime, @NonNegative long currentDuration) {

return TimeUnit.SECONDS.toNanos(3);

}

}).scheduler(Scheduler.systemScheduler())

.build();

cache.put(1, 2);

System.out.println(cache.getIfPresent(1));

Thread.sleep(6000);

System.out.println(cache.getIfPresent(1));

}

}

上面举了三个demo，已经是很详细了，这里需要额外提的一点是，我构建Cache对象的时候都会调用scheduler(Scheduler.systemScheduler())，Scheduler在上文描述Caffeine结构的时候有提到，Scheduler就是定期清空数据的一个机制，可以不设置，如果不设置则不会主动的清空过期数据。

实习生小张：diao大的稀饭，问题来了，如果不设置的时候数据过期了是什么时候清空的呢？

为了找出这个问题的答案，我特地通读了Caffeine的源码，终于找到答案，那就是在我们操作数据的时候会进行异步清空过期数据，也就是put或者get的时候，关于源码部分，等后面讲解完具体用法了我再特地讲讲。

实习生小张：还有一个问题，为什么我在我的工程中用了scheduler(Scheduler.systemScheduler())没生效呢？

这确实是一个很好的问题，如果没有像我这么仔细去看文档和跑demo的话根本不知道怎么解答这个问题，实际上是jdk版本的限制，只有java9以上才会生效。

**实际应用：目前在我司项目中，利用了基于缓存大小和访问后到期两种淘汰，目前从线上表现来说，效果是极其明显的，不过要注意一个点，那就是需要入库的缓存记得保存，否则容易产生数据丢失**。

最后一种淘汰机制是基于引用，很多人可能对引用没什么概念，在这里放过如意门：[https://mp.weixin.qq.com/s/-N...](https://link.segmentfault.com/?enc=kX3E%2FOy5LNTvtghl36ZZqg%3D%3D.tzuGxe6y8LI0uan%2FKyEa083T9WfhBYZFTiAKeYZVlvHKcPOsz789N592mzO67poMDJN1ECjbpwTpamylneZRCg%3D%3D" \t "_blank) ，不懂的先学学，再来看这个Caffeine吧。

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/25 0:43

\* @desc 基于引用淘汰

\*/

public class CaffeineReferenceEvictionTest {

@Test

public void testWeak() {

Cache<Integer, Integer> cache = Caffeine.newBuilder()

// 设置Key为弱引用，生命周期是下次gc的时候

.weakKeys()

// 设置value为弱引用，生命周期是下次gc的时候

.weakValues()

.build();

cache.put(1, 2);

System.out.println(cache.getIfPresent(1));

// 强行调用一次GC

System.gc();

System.out.println(cache.getIfPresent(1));

}

@Test

public void testSoft() {

Cache<Integer, Integer> cache = Caffeine.newBuilder()

// 设置value为软引用，生命周期是GC时并且堆内存不够时触发清除

.softValues()

.build();

cache.put(1, 2);

System.out.println(cache.getIfPresent(1));

// 强行调用一次GC

System.gc();

System.out.println(cache.getIfPresent(1));

}

}

这里要注意的地方有三个

* System.gc() 不一定会真的触发GC，只是一种通知机制，但是并非一定会发生GC，垃圾收集器进不进行GC是不确定的，所以有概率看到设置weakKeys了却在调用System.gc() 的时候却没有丢失缓存数据的情况。
* 使用异步加载的方式不允许使用引用淘汰机制，启动程序的时候会报错：java.lang.IllegalStateException: Weak or soft values can not be combined with AsyncCache，猜测原因是异步加载数据的生命周期和引用淘汰机制的生命周期冲突导致的，因而Caffeine不支持。
* 使用引用淘汰机制的时候，判断两个key或者两个value是否相同，用的是 ==，而非是equals()，也就是说需要两个key指向同一个对象才能被认为是一致的，这样极可能导致缓存命中出现预料之外的问题。

因而，总结下来就是慎用基于引用的淘汰机制，其实其他的淘汰机制完全够用了。

实习生小张：我这边接到了一个需求，需要用到写后一段时间定时过期，可是如果在一定时间内，数据有访问则重新计时，应该怎么做呢？

关于这种需求，其实并不常见，合理来说使用读写后过期真的够用，但是不排除有上面这种特别的情况。

因而这就要说到Caffeine提供的刷新机制了，使用很简单，用接口refreshAfterWrite 即可，可以说refreshAfterWrite其实就是和expireAfterWrite配套使用的，只不过使用refreshAfterWrite需要注意几个坑点，具体我举例说。

/\*\*

\* @author xifanxiaxue

\* @date 2020/12/1 23:12

\* @desc

\*/

public class CaffeineRefreshTest {

private int index = 1;

/\*\*

\* 模拟从数据库中读取数据

\*

\* @return

\*/

private int getInDB() {

// 这里为了体现数据重新被get，因而用了index++

index++;

return index;

}

@Test

public void test() throws InterruptedException {

// 设置写入后3秒后数据过期，2秒后如果有数据访问则刷新数据

LoadingCache<Integer, Integer> cache = Caffeine.newBuilder()

.refreshAfterWrite(2, TimeUnit.SECONDS)

.expireAfterWrite(3, TimeUnit.SECONDS)

.build(new CacheLoader<Integer, Integer>() {

@Nullable

@Override

public Integer load(@NonNull Integer key) {

return getInDB();

}

});

cache.put(1, getInDB());

// 休眠2.5秒，后取值

Thread.sleep(2500);

System.out.println(cache.getIfPresent(1));

// 休眠1.5秒，后取值

Thread.sleep(1500);

System.out.println(cache.getIfPresent(1));

}

}

可以看到我设置写入后3秒后数据过期，2秒后如果有数据访问则刷新数据，而在put数据后，我先是休眠了2.5秒，再打印取值，再休眠了1.5秒，再打印取值。

稀饭：小张，你猜猜看，最终打印是啥？

实习生小张：应该是 3 4，因为设置的写后刷新时间是2秒，而第一次休眠已经过了2.5秒了，应该已经主动打印了。

稀饭：其实不是，最终打印的结果是：2 3

**坑点：**我研究过源码，写后刷新其实并不是方法名描述的那样在一定时间后自动刷新，而是在一定时间后进行了访问，再访问后才自动刷新。也就是在第一次*cache.get(1)*的时候其实取到的依旧是旧值，在*doAfterRead*里边做了自动刷新的操作，这样在第二次*cache.get(1)*取到的才是刷洗后的值。

稀饭：那小张，你说说看，第一次休眠已经过了2.5秒，第二次休眠过了1.5秒，总共时长是4秒，而写后过期时间其实才设置了3秒，为什么第二次取值依旧取得到没有过期呢？

实习生小张：应该是这样的，在写后刷新后重新将值填充到了缓存中，因而触发了写后过期时间机制的重新计算，所以虽然看起来在第二次get数据的时候已经过了4秒，其实对于写后过期机制来说，其实才过了1.5秒。

稀饭：正解。

**坑点：**在写后刷新被触发后，会重新填充数据，因而会触发写后过期时间机制的重新计算。

实习生小张：主管说直接给Caffeine设置了最大缓存个数，会存在一个隐患，那便是当同时在线的玩家数超过最大缓存个数的情况下，会导致缓存被清，之后导致频繁读取数据库加载数据，让我在Caffeine的基础上，结合二级缓存解决这个问题。

可以的，目前来说Caffeine提供了整套机制，可以方便我们和二级缓存进行结合。

在具体给出例子前，要先引出一个CacheWriter的概念，我们可以把它当做一个回调对象，在往Caffeine的缓存put数据或者remove数据的时候回调用。

/\*\*

\* @author xifanxiaxue

\* @date 2020/12/5 10:18

\* @desc

\*/

public class CaffeineWriterTest {

/\*\*

\* 充当二级缓存用，生命周期仅活到下个gc

\*/

private Map<Integer, WeakReference<Integer>> secondCacheMap =

new ConcurrentHashMap<>();

@Test

public void test() throws InterruptedException {

// 设置最大缓存个数为1

LoadingCache<Integer, Integer> cache = Caffeine.newBuilder()

.maximumSize(1)

// 设置put和remove的回调

.writer(new CacheWriter<Integer, Integer>() {

@Override

public void write(@NonNull Integer key, @NonNull Integer value) {

secondCacheMap.put(key, new WeakReference<>(value));

System.out.println("触发CacheWriter.write，将key = " + key + "放入二级缓存中");

}

@Override

public void delete(@NonNull Integer key, @Nullable Integer value, @NonNull RemovalCause cause) {

switch (cause) {

case EXPLICIT:

secondCacheMap.remove(key);

System.out.println("触发CacheWriter" +

".delete，清除原因：主动清除，将key = " + key +

"从二级缓存清除");

break;

case SIZE:

System.out.println("触发CacheWriter" +

".delete，清除原因：缓存个数超过上限，key = " + key);

break;

default:

break;

}

}

})

.build(new CacheLoader<Integer, Integer>() {

@Nullable

@Override

public Integer load(@NonNull Integer key) {

WeakReference<Integer> value = secondCacheMap.get(key);

if (value == null) {

return null;

}

System.out.println("触发CacheLoader.load，从二级缓存读取key = " + key);

return value.get();

}

});

cache.put(1, 1);

cache.put(2, 2);

// 由于清除缓存是异步的，因而睡眠1秒等待清除完成

Thread.sleep(1000);

// 缓存超上限触发清除后

System.out.println("从Caffeine中get数据，key为1，value为"+cache.get(1));

}

}

举的这个例子稍显复杂，毕竟是要和二级缓存结合使用，不复杂点就没办法显示Caffeine的妙，先看下secondCacheMap对象，这是我用来充当二级缓存用的，由于value值我设置成为WeakReference弱引用，因而生命周期仅活到下个gc。

稀饭：小张，这个例子就可以解决你的二级缓存如何结合的问题，你给我说说看最终打印结果值是null还是非null？

小张：肯定是null 啊，因为key为1的缓存因为缓存个数超过上限被清除了呀。

对Caffeine的运行机制不够熟悉的人很容易犯了小张这样的错误，产生了对结果的误判。

为了理清楚程序运行的逻辑，我将程序运行结果打印了出来

触发CacheWriter.write，将key = 1放入二级缓存中  
触发CacheWriter.write，将key = 2放入二级缓存中  
触发CacheWriter.delete，清除原因：缓存个数超过上限，key = 1  
触发CacheLoader.load，从二级缓存读取key = 1  
从Caffeine中get数据，key为1，value为1  
触发CacheWriter.delete，清除原因：缓存个数超过上限，key = 2

结合代码，我们可以看到CacheWriter.delete中，我判断了RemovalCause，也就是清除缓存理由的意思，如果是缓存超上限，那么并不清除二级缓存的数据，而CacheLoader.load会从二级缓存中读取数据，所以在最终从Caffeine中加载key为1的数据的时候并不为null，而是从二级缓存拿到了数据。

实习生小张：那最后的打印 ” 触发CacheWriter.delete，清除原因：缓存个数超过上限，key = 2 “ 又是什么情况呢？

那是因为Caffeine在调用*CacheLoader.load*拿到非null的数据后会重新放入缓存中，这样便导致缓存个数又超过了最大的上限了，所以清除了key为2的缓存。

实习生小张：稀饭稀饭，我这边想具体看到缓存命中率如何，有没有什么方法呢？

有的有的，看源码的话就可以看到，Caffeine内部有挺多打点记录的，不过需要我们在构建缓存的时候开启记录。

/\*\*

\* @author xifanxiaxue

\* @date 2020/12/1 23:12

\* @desc

\*/

public class CaffeineRecordTest {

/\*\*

\* 模拟从数据库中读取数据

\*

\* @param key

\* @return

\*/

private int getInDB(int key) {

return key;

}

@Test

public void test() {

LoadingCache<Integer, Integer> cache = Caffeine.newBuilder()

// 开启记录

.recordStats()

.build(new CacheLoader<Integer, Integer>() {

@Override

public @Nullable Integer load(@NonNull Integer key) {

return getInDB(key);

}

});

cache.get(1);

// 命中率

System.out.println(cache.stats().hitRate());

// 被剔除的数量

System.out.println(cache.stats().evictionCount());

// 加载新值所花费的平均时间[纳秒]

System.out.println(cache.stats().averageLoadPenalty() );

}

}

**实际应用：**上次在游戏中引入Caffeine的时候便用来*record*的机制，只不过是在测试的时候用，一般不建议生产环境用这个。具体用法我是开了条线程定时的打印命中率、被剔除的数量以及加载新值所花费的平均时间，进而判断引入Caffeine是否具备一定的价值。

实习生小张：Caffeine我已经用上了，可是会有个问题，如果数据忘记保存入库，然后被淘汰掉了，玩家数据就丢失了，Caffeine有没有提供什么方法可以在淘汰的事情让开发者做点什么？

稀饭：还真的有，Caffeine对外提供了淘汰监听，我们只需要在监听器内进行保存就可以了。

/\*\*

\* @author xifanxiaxue

\* @date 2020/11/19 22:34

\* @desc 淘汰通知

\*/

public class CaffeineRemovalListenerTest {

@Test

public void test() throws InterruptedException {

LoadingCache<Integer, Integer> cache = Caffeine.newBuilder()

.expireAfterAccess(1, TimeUnit.SECONDS)

.scheduler(Scheduler.systemScheduler())

// 增加了淘汰监听

.removalListener(((key, value, cause) -> {

System.out.println("淘汰通知，key：" + key + "，原因：" + cause);

}))

.build(new CacheLoader<Integer, Integer>() {

@Override

public @Nullable Integer load(@NonNull Integer key) throws Exception {

return key;

}

});

cache.put(1, 2);

Thread.currentThread().sleep(2000);

}

可以看到我这边使用removalListener提供了淘汰监听，因此可以看到以下的打印结果：

淘汰通知，key：1，原因：EXPIRED

实习生小张：我看到数据淘汰的时候是有提供了几个cause的，也就是原因，分别对应着什么呢？

目前数据被淘汰的原因不外有以下几个：

* EXPLICIT：如果原因是这个，那么意味着数据被我们手动的remove掉了。
* REPLACED：就是替换了，也就是put数据的时候旧的数据被覆盖导致的移除。
* COLLECTED：这个有歧义点，其实就是收集，也就是垃圾回收导致的，一般是用弱引用或者软引用会导致这个情况。
* EXPIRED：数据过期，无需解释的原因。
* SIZE：个数超过限制导致的移除。

以上这几个就是数据淘汰时会携带的原因了，如果有需要，我们可以根据不同的原因处理不同的业务。

**实际应用：** 目前我们项目中就有在数据被淘汰的时候做了缓存入库的处理，毕竟确实有开发人员忘记在逻辑处理完后手动save入库了，所以只能做个兜底机制，避免数据丢失。

目前Caffeine系列教程基本就这样了，后续准备分享一个Caffeine结合db定时持久化的组件，有兴趣关注公众号，带你看看各种有用的技术。