**本地缓存Guava Cache以及Caffeine使用**

当前版本号：[v0.1]

最后修改时间：[2020/7/10]

* 文件版本说明

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 发布时间 | 修订章节 | 作者 |
| v0.1 | 2020/7/10 | 新建文档 | 熊文辉 |
|  |  |  |  |
|  |  |  |  |

* 参考资料

1、https://github.com/google/guava

2、https://github.com/ben-manes/caffeine

* 手册目的

记录本地缓存Guava Cache以及Caffeine使用方法和区别

* 声明
* 名词定义和缩略语说明

| 序号 | 缩写 | 说明 |
| --- | --- | --- |
| 1 | Starry | 业务数据查询项目 |
| 3 |  |  |

# 一、Guava Cache

Guava Cache是在内存中缓存数据，相比较于数据库或redis存储，访问内存中的数据会更加高效。Guava官网介绍，下面的这几种情况可以考虑使用Guava Cache：

1、愿意消耗一些内存空间来提升速度。

2、预料到某些键会被多次查询。

3、缓存中存放的数据总量不会超出内存容量。

所以，可以将程序频繁用到的少量数据存储到Guava Cache中，以改善程序性能。下面对Guava Cache的用法进行详细的介绍。

## 1、构建缓存对象

接口Cache代表一块缓存，它有如下方法：

public interface Cache<K, V> {

V get(K key, Callable<? extends V> valueLoader) throws ExecutionException;

ImmutableMap<K, V> getAllPresent(Iterable<?> keys);

void put(K key, V value);

void putAll(Map<? extends K, ? extends V> m);

void invalidate(Object key);

void invalidateAll(Iterable<?> keys);

void invalidateAll();

long size();

CacheStats stats();

ConcurrentMap<K, V> asMap();

void cleanUp();

}

可以通过CacheBuilder类构建一个缓存对象，CacheBuilder类采用builder设计模式，它的每个方法都返回CacheBuilder本身，直到build方法被调用。构建一个缓存对象代码如下。

public class StudyGuavaCache {

public static void main(String[] args) {

Cache<String,String> cache = CacheBuilder.newBuilder().build();

cache.put("word","Hello Guava Cache");

System.out.println(cache.getIfPresent("word"));

}

}

上面的代码通过CacheBuilder.newBuilder().build()这句代码创建了一个Cache缓存对象，并在缓存对象中存储了key为word，value为Hello Guava Cache的一条记录。可以看到Cache非常类似于JDK中的Map，但是相比于Map，Guava Cache提供了很多更强大的功能。

## 2、设置最大存储

Guava Cache可以在构建缓存对象时指定缓存所能够存储的最大记录数量。当Cache中的记录数量达到最大值后再调用put方法向其中添加对象，Guava会先从当前缓存的对象记录中选择一条删除掉，腾出空间后再将新的对象存储到Cache中。

public class StudyGuavaCache {

public static void main(String[] args) {

Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(2)

.build();

cache.put("key1","value1");

cache.put("key2","value2");

cache.put("key3","value3");

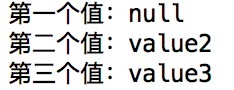
System.out.println("第一个值：" + cache.getIfPresent("key1"));

System.out.println("第二个值：" + cache.getIfPresent("key2"));

System.out.println("第三个值：" + cache.getIfPresent("key3"));

}

}

上面代码在构造缓存对象时，通过CacheBuilder类的maximumSize方法指定Cache最多可以存储两个对象，然后调用Cache的put方法向其中添加了三个对象。程序执行结果如下图所示，可以看到第三条对象记录的插入，导致了第一条对象记录被删除。  


## 3、设置过期时间

在构建Cache对象时，可以通过CacheBuilder类的expireAfterAccess和expireAfterWrite两个方法为缓存中的对象指定过期时间，过期的对象将会被缓存自动删除。其中，expireAfterWrite方法指定对象被写入到缓存后多久过期，expireAfterAccess指定对象多久没有被访问后过期。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(2)

.expireAfterWrite(3,TimeUnit.SECONDS)

.build();

cache.put("key1","value1");

int time = 1;

while(true) {

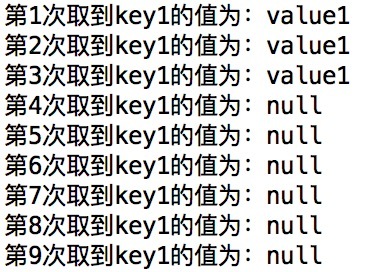
System.out.println("第" + time++ + "次取到key1的值为：" + cache.getIfPresent("key1"));

Thread.sleep(1000);

}

}

}

上面的代码在构造Cache对象时，通过CacheBuilder的expireAfterWrite方法指定put到Cache中的对象在3秒后会过期。在Cache对象中存储一条对象记录后，每隔1秒读取一次这条记录。程序运行结果如下图所示，可以看到，前三秒可以从Cache中获取到对象，超过三秒后，对象从Cache中被自动删除。  


下面代码是expireAfterAccess的例子。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(2)

.expireAfterAccess(3,TimeUnit.SECONDS)

.build();

cache.put("key1","value1");

int time = 1;

while(true) {

Thread.sleep(time\*1000);

System.out.println("睡眠" + time++ + "秒后取到key1的值为：" + cache.getIfPresent("key1"));

}

}

}

通过CacheBuilder的expireAfterAccess方法指定Cache中存储的对象如果超过3秒没有被访问就会过期。while中的代码每sleep一段时间就会访问一次Cache中存储的对象key1，每次访问key1之后下次sleep的时间会加长一秒。程序运行结果如下图所示，从结果中可以看出，当超过3秒没有读取key1对象之后，该对象会自动被Cache删除。  


也可以同时用expireAfterAccess和expireAfterWrite方法指定过期时间，这时只要对象满足两者中的一个条件就会被自动过期删除。

## 4、弱引用

可以通过weakKeys和weakValues方法指定Cache只保存对缓存记录key和value的弱引用。这样当没有其他强引用指向key和value时，key和value对象就会被垃圾回收器回收。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

Cache<String,Object> cache = CacheBuilder.newBuilder()

.maximumSize(2)

.weakValues()

.build();

Object value = new Object();

cache.put("key1",value);

value = new Object();//原对象不再有强引用

System.gc();

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

}

}

上面代码的打印结果是null。构建Cache时通过weakValues方法指定Cache只保存记录值的一个弱引用。当给value引用赋值一个新的对象之后，就不再有任何一个强引用指向原对象。System.gc()触发垃圾回收后，原对象就被清除了。

## 5、显示清除

可以调用Cache的invalidateAll或invalidate方法显示删除Cache中的记录。invalidate方法一次只能删除Cache中一个记录，接收的参数是要删除记录的key。invalidateAll方法可以批量删除Cache中的记录，当没有传任何参数时，invalidateAll方法将清除Cache中的全部记录。invalidateAll也可以接收一个Iterable类型的参数，参数中包含要删除记录的所有key值。下面代码对此做了示例。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

Cache<String,String> cache = CacheBuilder.newBuilder().build();

Object value = new Object();

cache.put("key1","value1");

cache.put("key2","value2");

cache.put("key3","value3");

List<String> list = new ArrayList<String>();

list.add("key1");

list.add("key2");

cache.invalidateAll(list);//批量清除list中全部key对应的记录

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

System.out.println(cache.getIfPresent("key2"));

System.out.println(cache.getIfPresent("key3"));

}

}

代码中构造了一个集合list用于保存要删除记录的key值，然后调用invalidateAll方法批量删除key1和key2对应的记录，只剩下key3对应的记录没有被删除。

## 6、移除监听器

可以为Cache对象添加一个移除监听器，这样当有记录被删除时可以感知到这个事件。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

RemovalListener<String, String> listener = new RemovalListener<String, String>() {

public void onRemoval(RemovalNotification<String, String> notification) {

System.out.println("[" + notification.getKey() + ":" + notification.getValue() + "] is removed!");

}

};

Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(3)

.removalListener(listener)

.build();

Object value = new Object();

cache.put("key1","value1");

cache.put("key2","value2");

cache.put("key3","value3");

cache.put("key4","value3");

cache.put("key5","value3");

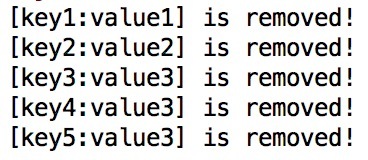
cache.put("key6","value3");

cache.put("key7","value3");

cache.put("key8","value3");

}

}

removalListener方法为Cache指定了一个移除监听器，这样当有记录从Cache中被删除时，监听器listener就会感知到这个事件。程序运行结果如下图所示。  


## 7、自动加载

Cache的get方法有两个参数，第一个参数是要从Cache中获取记录的key，第二个记录是一个Callable对象。当缓存中已经存在key对应的记录时，get方法直接返回key对应的记录。如果缓存中不包含key对应的记录，Guava会启动一个线程执行Callable对象中的call方法，call方法的返回值会作为key对应的值被存储到缓存中，并且被get方法返回。下面是一个多线程的例子：

public class StudyGuavaCache {

private static Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(3)

.build();

public static void main(String[] args) throws InterruptedException {

new Thread(new Runnable() {

public void run() {

System.out.println("thread1");

try {

String value = cache.get("key", new Callable<String>() {

public String call() throws Exception {

System.out.println("load1"); //加载数据线程执行标志

Thread.sleep(1000); //模拟加载时间

return "auto load by Callable";

}

});

System.out.println("thread1 " + value);

} catch (ExecutionException e) {

e.printStackTrace();

}

}

}).start();

new Thread(new Runnable() {

public void run() {

System.out.println("thread2");

try {

String value = cache.get("key", new Callable<String>() {

public String call() throws Exception {

System.out.println("load2"); //加载数据线程执行标志

Thread.sleep(1000); //模拟加载时间

return "auto load by Callable";

}

});

System.out.println("thread2 " + value);

} catch (ExecutionException e) {

e.printStackTrace();

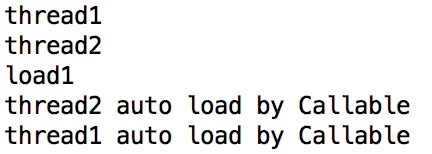
}

}

}).start();

}

}

这段代码中有两个线程共享同一个Cache对象，两个线程同时调用get方法获取同一个key对应的记录。由于key对应的记录不存在，所以两个线程都在get方法处阻塞。此处在call方法中调用Thread.sleep(1000)模拟程序从外存加载数据的时间消耗。代码的执行结果如下图：  


从结果中可以看出，虽然是两个线程同时调用get方法，但只有一个get方法中的Callable会被执行(没有打印出load2)。Guava可以保证当有多个线程同时访问Cache中的一个key时，如果key对应的记录不存在，Guava只会启动一个线程执行get方法中Callable参数对应的任务加载数据存到缓存。当加载完数据后，任何线程中的get方法都会获取到key对应的值。

## 8、统计信息

可以对Cache的命中率、加载数据时间等信息进行统计。在构建Cache对象时，可以通过CacheBuilder的recordStats方法开启统计信息的开关。开关开启后Cache会自动对缓存的各种操作进行统计，调用Cache的stats方法可以查看统计后的信息。

public class StudyGuavaCache {

public static void main(String[] args) throws InterruptedException {

Cache<String,String> cache = CacheBuilder.newBuilder()

.maximumSize(3)

.recordStats() //开启统计信息开关

.build();

cache.put("key1","value1");

cache.put("key2","value2");

cache.put("key3","value3");

cache.put("key4","value4");

cache.getIfPresent("key1");

cache.getIfPresent("key2");

cache.getIfPresent("key3");

cache.getIfPresent("key4");

cache.getIfPresent("key5");

cache.getIfPresent("key6");

System.out.println(cache.stats()); //获取统计信息

}

}

程序执行结果如下图所示：  
IMG_256

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应该密切关注这些数据

## 9、LoadingCache

LoadingCache是Cache的子接口，相比较于Cache，当从LoadingCache中读取一个指定key的记录时，如果该记录不存在，则LoadingCache可以自动执行加载数据到缓存的操作。LoadingCache接口的定义如下：

public interface LoadingCache<K, V> extends Cache<K, V>, Function<K, V> {

V get(K key) throws ExecutionException;

V getUnchecked(K key);

ImmutableMap<K, V> getAll(Iterable<? extends K> keys) throws ExecutionException;

V apply(K key);

void refresh(K key);

@Override

ConcurrentMap<K, V> asMap();

}

与构建Cache类型的对象类似，LoadingCache类型的对象也是通过CacheBuilder进行构建，不同的是，在调用CacheBuilder的build方法时，必须传递一个CacheLoader类型的参数，CacheLoader的load方法需要我们提供实现。当调用LoadingCache的get方法时，如果缓存不存在对应key的记录，则CacheLoader中的load方法会被自动调用从外存加载数据，load方法的返回值会作为key对应的value存储到LoadingCache中，并从get方法返回。

public class StudyGuavaCache {

public static void main(String[] args) throws ExecutionException {

CacheLoader<String, String> loader = new CacheLoader<String, String> () {

public String load(String key) throws Exception {

Thread.sleep(1000); //休眠1s，模拟加载数据

System.out.println(key + " is loaded from a cacheLoader!");

return key + "'s value";

}

};

LoadingCache<String,String> loadingCache = CacheBuilder.newBuilder()

.maximumSize(3)

.build(loader);//在构建时指定自动加载器

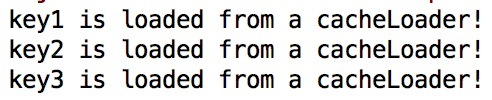
loadingCache.get("key1");

loadingCache.get("key2");

loadingCache.get("key3");

}

}

程序执行结果如下图所示：  


# 二、Caffeine Cache

Caffeine是新出现的一个高性能的Java缓存，有了它完全可以代替Guava Cache，来实现更加高效的缓存；Caffeine采用了W-TinyLFU回收策略，集合了LRU和LFU的优点，提供了一个最佳的命中率。

## 1、缓存填充策略

Caffeine Cache提供了三种缓存填充策略：手动、同步加载和异步加载。

### 1.1、手动加载

在每次get key的时候指定一个同步的函数，如果key不存在就调用这个函数生成一个值。

/\*\*

\* 手动加载

\* @param key

\* @return

\*/

public Object manulOperator(String key) {

Cache<String, Object> cache = Caffeine.newBuilder()

.expireAfterWrite(1, TimeUnit.SECONDS)

.expireAfterAccess(1, TimeUnit.SECONDS)

.maximumSize(10)

.build();

//如果一个key不存在，那么会进入指定的函数生成value

Object value = cache.get(key, t -> setValue(key).apply(key));

cache.put("hello",value);

//判断是否存在如果不存返回null

Object ifPresent = cache.getIfPresent(key);

//移除一个key

cache.invalidate(key);

return value;

}

public Function<String, Object> setValue(String key){

return t -> key + "value";

}

### 1.2、同步加载

构造Cache时候，build方法传入一个CacheLoader实现类。实现load方法，通过key加载value。

/\*\*

\* 同步加载

\* @param key

\* @return

\*/

public Object syncOperator(String key){

LoadingCache<String, Object> cache = Caffeine.newBuilder()

.maximumSize(100)

.expireAfterWrite(1, TimeUnit.MINUTES)

.build(k -> setValue(key).apply(key));

return cache.get(key);

}

public Function<String, Object> setValue(String key){

return t -> key + "value";

}

### 1.3、异步加载

AsyncLoadingCache是继承自LoadingCache类的，异步加载使用Executor去调用方法并返回一个CompletableFuture。异步加载缓存使用了响应式编程模型。

如果要以同步方式调用时，应提供CacheLoader。要以异步表示时，应该提供一个AsyncCacheLoader，并返回一个CompletableFuture。

/\*\*

\* 异步加载

\*

\* @param key

\* @return

\*/

public Object asyncOperator(String key){

AsyncLoadingCache<String, Object> cache = Caffeine.newBuilder()

.maximumSize(100)

.expireAfterWrite(1, TimeUnit.MINUTES)

.buildAsync(k -> setAsyncValue(key).get());

return cache.get(key);

}

public CompletableFuture<Object> setAsyncValue(String key){

return CompletableFuture.supplyAsync(() -> {

return key + "value";

});

}

## 2、回收策略

Caffeine提供了3种回收策略：基于大小回收，基于时间回收，基于引用回收。

### 2.1、基于大小的过期方式

基于大小的回收策略有两种方式：一种是基于缓存大小，一种是基于权重。

// 根据缓存的计数进行驱逐

LoadingCache<String, Object> cache = Caffeine.newBuilder()

.maximumSize(10000)

.build(key -> function(key));

// 根据缓存的权重来进行驱逐（权重只是用于确定缓存大小，不会用于决定该缓存是否被驱逐）

LoadingCache<String, Object> cache1 = Caffeine.newBuilder()

.maximumWeight(10000)

.weigher(key -> function1(key))

.build(key -> function(key));

maximumWeight与maximumSize不可以同时使用。

### 2.2、基于时间的过期方式

// 基于固定的到期策略进行退出

LoadingCache<String, Object> cache = Caffeine.newBuilder()

.expireAfterAccess(5, TimeUnit.MINUTES)

.build(key -> function(key));

LoadingCache<String, Object> cache1 = Caffeine.newBuilder()

.expireAfterWrite(10, TimeUnit.MINUTES)

.build(key -> function(key));

// 基于不同的到期策略进行退出

LoadingCache<String, Object> cache2 = Caffeine.newBuilder()

.expireAfter(new Expiry<String, Object>() {

@Override

public long expireAfterCreate(String key, Object value, long currentTime) {

return TimeUnit.SECONDS.toNanos(seconds);

}

@Override

public long expireAfterUpdate(@Nonnull String s, @Nonnull Object o, long l, long l1) {

return 0;

}

@Override

public long expireAfterRead(@Nonnull String s, @Nonnull Object o, long l, long l1) {

return 0;

}

}).build(key -> function(key));

Caffeine提供了三种定时驱逐策略：

expireAfterAccess(long, TimeUnit):在最后一次访问或者写入后开始计时，在指定的时间后过期。假如一直有请求访问该key，那么这个缓存将一直不会过期。  
expireAfterWrite(long, TimeUnit): 在最后一次写入缓存后开始计时，在指定的时间后过期。  
expireAfter(Expiry): 自定义策略，过期时间由Expiry实现独自计算。  
缓存的删除策略使用的是惰性删除和定时删除。这两个删除策略的时间复杂度都是O(1)。

### 2.3、基于引用的过期方式

Java中四种引用类型

| 引用类型 | 被垃圾回收时间 | 用途 | 生存时间 |
| --- | --- | --- | --- |
| 强引用 Strong Reference | 从来不会 | 对象的一般状态 | JVM停止运行时终止 |
| 软引用 Soft Reference | 在内存不足时 | 对象缓存 | 内存不足时终止 |
| 弱引用 Weak Reference | 在垃圾回收时 | 对象缓存 | gc运行后终止 |
| 虚引用 Phantom Reference | 从来不会 | 可以用虚引用来跟踪对象被垃圾回收器回收的活动，当一个虚引用关联的对象被垃圾收集器回收之前会收到一条系统通知 | JVM停止运行时终止 |

// 当key和value都没有引用时驱逐缓存

LoadingCache<String, Object> cache = Caffeine.newBuilder()

.weakKeys()

.weakValues()

.build(key -> function(key));

// 当垃圾收集器需要释放内存时驱逐

LoadingCache<String, Object> cache1 = Caffeine.newBuilder()

.softValues()

.build(key -> function(key));

注意：AsyncLoadingCache不支持弱引用和软引用。

● Caffeine.weakKeys()： 使用弱引用存储key。如果没有其他地方对该key有强引用，那么该缓存就会被垃圾回收器回收。由于垃圾回收器只依赖于身份(identity)相等，因此这会导致整个缓存使用身份 (==) 相等来比较 key，而不是使用 equals()。

● Caffeine.weakValues() ：使用弱引用存储value。如果没有其他地方对该value有强引用，那么该缓存就会被垃圾回收器回收。由于垃圾回收器只依赖于身份(identity)相等，因此这会导致整个缓存使用身份 (==) 相等来比较 key，而不是使用 equals()。

● Caffeine.softValues() ：使用软引用存储value。当内存满了过后，软引用的对象以将使用最近最少使用(least-recently-used ) 的方式进行垃圾回收。由于使用软引用是需要等到内存满了才进行回收，所以我们通常建议给缓存配置一个使用内存的最大值。 softValues() 将使用身份相等(identity) (==) 而不是equals() 来比较值。

Caffeine.weakValues()和Caffeine.softValues()不可以一起使用。

## 3、移除事件监听

Cache<String, Object> cache = Caffeine.newBuilder()

.removalListener((String key, Object value, RemovalCause cause) ->

System.out.printf("Key %s was removed (%s)%n", key, cause))

.build();

## 4、写入外部存储

CacheWriter 方法可以将缓存中所有的数据写入到第三方。

LoadingCache<String, Object> cache2 = Caffeine.newBuilder()

.writer(new CacheWriter<String, Object>() {

@Override public void write(String key, Object value) {

// 写入到外部存储

}

@Override public void delete(String key, Object value, RemovalCause cause) {

// 删除外部存储

}

})

.build(key -> function(key));

如果你有多级缓存的情况下，这个方法还是很实用。

注意：CacheWriter不能与弱键或AsyncLoadingCache一起使用。

## 5、统计

与Guava Cache的统计一样。

Cache<String, Object> cache = Caffeine.newBuilder()

.maximumSize(10\_000)

.recordStats()

.build();

通过使用Caffeine.recordStats(), 可以转化成一个统计的集合. 通过 Cache.stats() 返回一个CacheStats。CacheStats提供以下统计方法：

● hitRate(): 返回缓存命中率

● evictionCount(): 缓存回收数量

● averageLoadPenalty(): 加载新值的平均时间

## 6、SpringBoot中默认Cache-Caffine Cache

SpringBoot 1.x版本中的默认本地cache是Guava Cache。在2.x（Spring Boot 2.0(spring 5) ）版本中已经用Caffine Cache取代了Guava Cache。毕竟有了更优的缓存淘汰策略。

下面我们来说在SpringBoot2.x版本中如何使用cache。

### 6.1、引入依赖：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

<dependency>

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

<artifactId>caffeine</artifactId>

<version>2.6.2</version>

</dependency>

### 6.2、添加注解开启缓存支持

添加@EnableCaching注解：

@SpringBootApplication@EnableCachingpublic class SingleDatabaseApplication {

public static void main(String[] args) {

SpringApplication.run(SingleDatabaseApplication.class, args);

}

}

### 6.3、配置文件的方式注入相关参数

properties文件：

spring.cache.cache-names=cache1

spring.cache.caffeine.spec=initialCapacity=50,maximumSize=500,expireAfterWrite=10s

或Yaml文件

spring:

cache:

type: caffeine

cache-names:

- userCache

caffeine:

spec: maximumSize=1024,refreshAfterWrite=60s

如果使用refreshAfterWrite配置,必须指定一个CacheLoader.不用该配置则无需这个bean,如上所述,该CacheLoader将关联被该缓存管理器管理的所有缓存，所以必须定义为CacheLoader<Object, Object>，自动配置将忽略所有泛型类型。

import com.github.benmanes.caffeine.cache.CacheLoader;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

/\*\*

\* @author: rickiyang

\* @date: 2019/6/15

\* @description:

\*/

@Configuration

public class CacheConfig {

/\*\*

\* 相当于在构建LoadingCache对象的时候 build()方法中指定过期之后的加载策略方法

\* 必须要指定这个Bean，refreshAfterWrite=60s属性才生效

\* @return

\*/

@Bean

public CacheLoader<String, Object> cacheLoader() {

CacheLoader<String, Object> cacheLoader = new CacheLoader<String, Object>() {

@Override

public Object load(String key) throws Exception {

return null;

}

// 重写这个方法将oldValue值返回回去，进而刷新缓存

@Override

public Object reload(String key, Object oldValue) throws Exception {

return oldValue;

}

};

return cacheLoader;

}

}

Caffeine常用配置说明：

● initialCapacity=[integer]: 初始的缓存空间大小

● maximumSize=[long]: 缓存的最大条数

● maximumWeight=[long]: 缓存的最大权重

● expireAfterAccess=[duration]: 最后一次写入或访问后经过固定时间过期

● expireAfterWrite=[duration]: 最后一次写入后经过固定时间过期

● refreshAfterWrite=[duration]: 创建缓存或者最近一次更新缓存后经过固定的时间间隔，刷新缓存

● weakKeys: 打开key的弱引用

● weakValues：打开value的弱引用

● softValues：打开value的软引用

● recordStats：开发统计功能

注意：

● expireAfterWrite和expireAfterAccess同时存在时，以expireAfterWrite为准。

● maximumSize和maximumWeight不可以同时使用

● weakValues和softValues不可以同时使用

需要说明的是，使用配置文件的方式来进行缓存项配置，一般情况能满足使用需求，但是灵活性不是很高，如果我们有很多缓存项的情况下写起来会导致配置文件很长。所以一般情况下你也可以选择使用bean的方式来初始化Cache实例。

下面的演示使用bean的方式来注入：

package com.rickiyang.learn.cache;

import com.github.benmanes.caffeine.cache.CacheLoader;

import com.github.benmanes.caffeine.cache.Caffeine;

import org.apache.commons.compress.utils.Lists;

import org.springframework.cache.CacheManager;

import org.springframework.cache.caffeine.CaffeineCache;

import org.springframework.cache.support.SimpleCacheManager;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.TimeUnit;

/\*\*

\* @author: rickiyang

\* @date: 2019/6/15

\* @description:

\*/

@Configuration

public class CacheConfig {

/\*\*

\* 创建基于Caffeine的Cache Manager

\* 初始化一些key存入

\* @return

\*/

@Bean

@Primary

public CacheManager caffeineCacheManager() {

SimpleCacheManager cacheManager = new SimpleCacheManager();

ArrayList<CaffeineCache> caches = Lists.newArrayList();

List<CacheBean> list = setCacheBean();

for(CacheBean cacheBean : list){

caches.add(new CaffeineCache(cacheBean.getKey(),

Caffeine.newBuilder().recordStats()

.expireAfterWrite(cacheBean.getTtl(), TimeUnit.SECONDS)

.maximumSize(cacheBean.getMaximumSize())

.build()));

}

cacheManager.setCaches(caches);

return cacheManager;

}

/\*\*

\* 初始化一些缓存的 key

\* @return

\*/

private List<CacheBean> setCacheBean(){

List<CacheBean> list = Lists.newArrayList();

CacheBean userCache = new CacheBean();

userCache.setKey("userCache");

userCache.setTtl(60);

userCache.setMaximumSize(10000);

CacheBean deptCache = new CacheBean();

deptCache.setKey("userCache");

deptCache.setTtl(60);

deptCache.setMaximumSize(10000);

list.add(userCache);

list.add(deptCache);

return list;

}

class CacheBean {

private String key;

private long ttl;

private long maximumSize;

public String getKey() {

return key;

}

public void setKey(String key) {

this.key = key;

}

public long getTtl() {

return ttl;

}

public void setTtl(long ttl) {

this.ttl = ttl;

}

public long getMaximumSize() {

return maximumSize;

}

public void setMaximumSize(long maximumSize) {

this.maximumSize = maximumSize;

}

}

}

创建了一个SimpleCacheManager作为Cache的管理对象，然后初始化了两个Cache对象，分别存储user，dept类型的缓存。当然构建Cache的参数设置我写的比较简单，你在使用的时候酌情根据需要配置参数。

### 6.4、使用注解来对cache增删改查

我们可以使用spring提供的 @Cacheable、@CachePut、@CacheEvict等注解来方便的使用caffeine缓存。

如果使用了多个cahce，比如redis、caffeine等，必须指定某一个CacheManage为@primary，在@Cacheable注解中没指定 cacheManager 则使用标记为primary的那个。

cache方面的注解主要有以下5个：

● @Cacheable 触发缓存入口（这里一般放在创建和获取的方法上，@Cacheable注解会先查询是否已经有缓存，有会使用缓存，没有则会执行方法并缓存）

● @CacheEvict 触发缓存的eviction（用于删除的方法上）

● @CachePut 更新缓存且不影响方法执行（用于修改的方法上，该注解下的方法始终会被执行）

● @Caching 将多个缓存组合在一个方法上（该注解可以允许一个方法同时设置多个注解）

● @CacheConfig 在类级别设置一些缓存相关的共同配置（与其它缓存配合使用）

说一下@Cacheable 和 @CachePut的区别：

● @Cacheable：它的注解的方法是否被执行取决于Cacheable中的条件，方法很多时候都可能不被执行。

● @CachePut：这个注解不会影响方法的执行，也就是说无论它配置的条件是什么，方法都会被执行，更多的时候是被用到修改上。

简要说一下Cacheable类中各个方法的使用：

public @interface Cacheable {

/\*\*

\* 要使用的cache的名字

\*/

@AliasFor("cacheNames")

String[] value() default {};

/\*\*

\* 同value()，决定要使用那个/些缓存

\*/

@AliasFor("value")

String[] cacheNames() default {};

/\*\*

\* 使用SpEL表达式来设定缓存的key，如果不设置默认方法上所有参数都会作为key的一部分

\*/

String key() default "";

/\*\*

\* 用来生成key，与key()不可以共用

\*/

String keyGenerator() default "";

/\*\*

\* 设定要使用的cacheManager，必须先设置好cacheManager的bean，这是使用该bean的名字

\*/

String cacheManager() default "";

/\*\*

\* 使用cacheResolver来设定使用的缓存，用法同cacheManager，但是与cacheManager不可以同时使用

\*/

String cacheResolver() default "";

/\*\*

\* 使用SpEL表达式设定出发缓存的条件，在方法执行前生效

\*/

String condition() default "";

/\*\*

\* 使用SpEL设置出发缓存的条件，这里是方法执行完生效，所以条件中可以有方法执行后的value

\*/

String unless() default "";

/\*\*

\* 用于同步的，在缓存失效（过期不存在等各种原因）的时候，如果多个线程同时访问被标注的方法

\* 则只允许一个线程通过去执行方法

\*/

boolean sync() default false;

}

基于注解的使用方法：

package com.rickiyang.learn.cache;

import com.rickiyang.learn.entity.User;

import org.springframework.cache.annotation.CacheEvict;

import org.springframework.cache.annotation.CachePut;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Service;

/\*\*

\* @author: rickiyang

\* @date: 2019/6/15

\* @description: 本地cache

\*/

@Service

public class UserCacheService {

/\*\*

\* 查找

\* 先查缓存，如果查不到，会查数据库并存入缓存

\* @param id

\*/

@Cacheable(value = "userCache", key = "#id", sync = true)

public void getUser(long id){

//查找数据库

}

/\*\*

\* 更新/保存

\* @param user

\*/

@CachePut(value = "userCache", key = "#user.id")

public void saveUser(User user){

//todo 保存数据库

}

/\*\*

\* 删除

\* @param user

\*/

@CacheEvict(value = "userCache",key = "#user.id")

public void delUser(User user){

//todo 保存数据库

}

}

如果你不想使用注解的方式去操作缓存，也可以直接使用SimpleCacheManager获取缓存的key进而进行操作。

注意到上面的key使用了spEL 表达式。Spring Cache提供了一些供我们使用的SpEL上下文数据，下表直接摘自Spring官方文档：

| 名称 | 位置 | 描述 | 示例 |
| --- | --- | --- | --- |
| methodName | root对象 | 当前被调用的方法名 | #root.methodname |
| method | root对象 | 当前被调用的方法 | #root.method.name |
| target | root对象 | 当前被调用的目标对象实例 | #root.target |
| targetClass | root对象 | 当前被调用的目标对象的类 | #root.targetClass |
| args | root对象 | 当前被调用的方法的参数列表 | #root.args[0] |
| caches | root对象 | 当前方法调用使用的缓存列表 | #root.caches[0].name |
| Argument Name | 执行上下文 | 当前被调用的方法的参数，如findArtisan(Artisan artisan),可以通过#artsian.id获得参数 | #artsian.id |
| result | 执行上下文 | 方法执行后的返回值（仅当方法执行后的判断有效，如 unless cacheEvict的beforeInvocation=false） | #result |

注意：

1、当我们要使用root对象的属性作为key时我们也可以将“#root”省略，因为Spring默认使用的就是root对象的属性。 如

@Cacheable(key = "targetClass + methodName +#p0")

2、使用方法参数时我们可以直接使用“#参数名”或者“#p参数index”。 如：

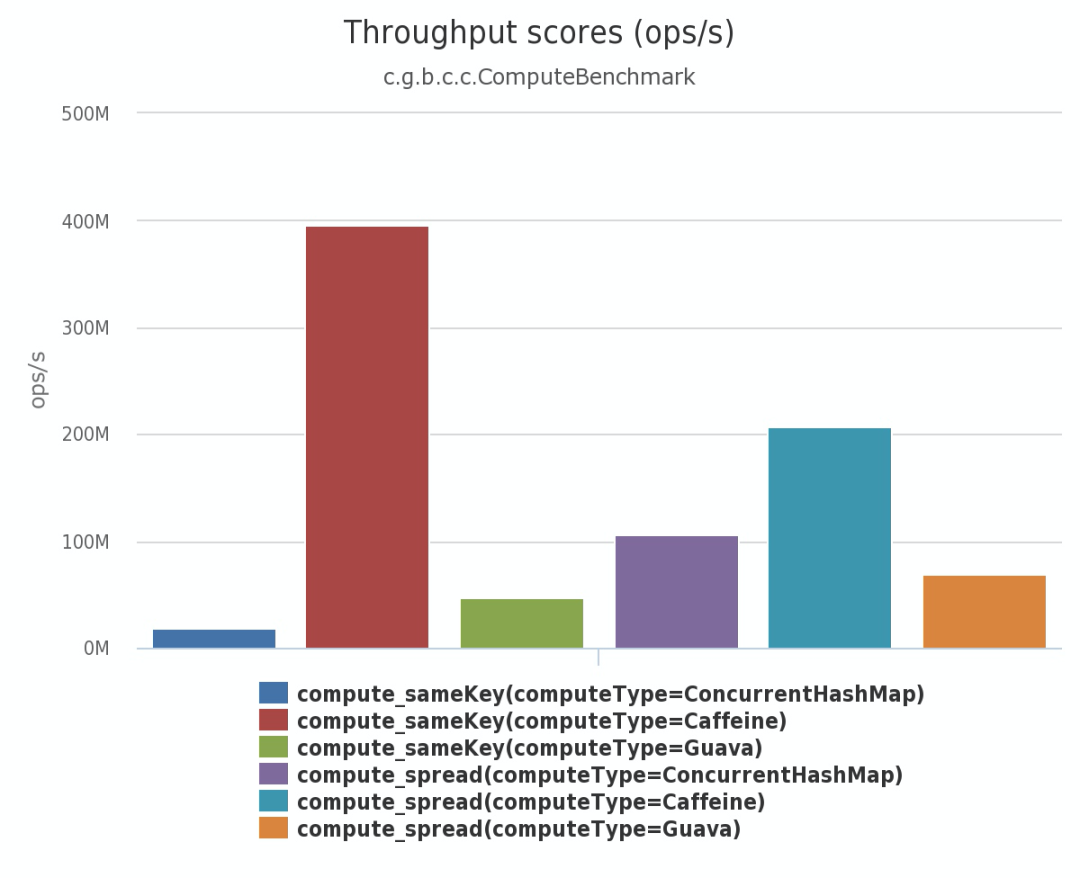
@Cacheable(value="userCache", key="#id")@Cacheable(value="userCache", key="#p0")

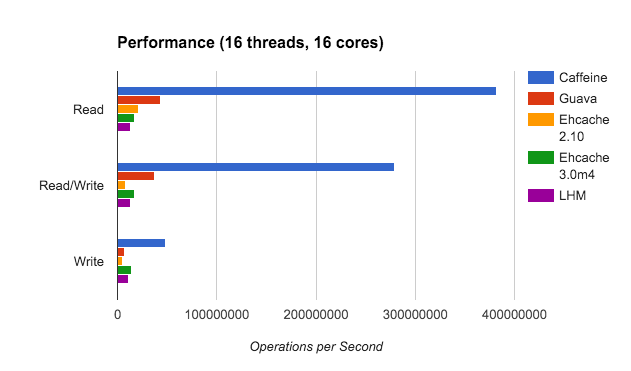
SpEL提供了多种运算符

| 类型 | 运算符 |
| --- | --- |
| 关系 | <，>，<=，>=，==，!=，lt，gt，le，ge，eq，ne |
| 算术 | +，- ，\* ，/，%，^ |
| 逻辑 | &&，||，!，and，or，not，between，instanceof |
| 条件 | ?: (ternary)，?: (elvis) |
| 正则表达式 | matches |
| 其他类型 | ?.，?[…]，![…]，^[…]，$[…] |

# 三、Guava Cache VS Caffeine Cache

## 1、性能比较





Caffeine支持异步加载方式，直接返回CompletableFutures，相对于GuavaCache的同步方式，它不用阻塞等待数据的载入，另外，它的编程模型是友好的，省去了很多重复的工作。

Caffeine采用了W-TinyLFU回收策略，集合了LRU和LFU的优点，提供了一个最佳的命中率，在效率上可以秒杀Guava Cache。

## 2、缓存淘汰算法

常见的缓存淘汰算法还有FIFO、LRU、LFU：

● FIFO：先进先出，在这种淘汰算法中，先进入缓存的会先被淘汰，会导致命中率很低。

● LRU：最近最少使用算法，每次访问数据都会将其放在我们的队尾，如果需要淘汰数据，就只需要淘汰队首即可。仍然有个问题，如果有个数据在 1 分钟访问了 1000次，再后 1 分钟没有访问这个数据，但是有其他的数据访问，就导致了我们这个热点数据被淘汰。

● LFU：最近最少频率使用，利用额外的空间记录每个数据的使用频率，然后选出频率最低进行淘汰。这样就避免了 LRU 不能处理时间段的问题。

上面三种策略各有利弊，实现的成本也是一个比一个高，同时命中率也是一个比一个好。Guava Cache虽然有这么多的功能，但是本质上还是对LRU的封装，如果有更优良的算法，并且也能提供这么多功能，相比之下就相形见绌了。

LFU的局限性：在 LFU 中只要数据访问模式的概率分布随时间保持不变时，其命中率就能变得非常高。比如有部新剧出来了，我们使用 LFU 给他缓存下来，这部新剧在这几天大概访问了几亿次，这个访问频率也在我们的 LFU 中记录了几亿次。但是新剧总会过气的，比如一个月之后这个新剧的前几集其实已经过气了，但是他的访问量的确是太高了，其他的电视剧根本无法淘汰这个新剧，所以在这种模式下是有局限性。

LRU的优点和局限性：LRU可以很好的应对突发流量的情况，因为他不需要累计数据频率。但LRU通过历史数据来预测未来是局限的，它会认为最后到来的数据是最可能被再次访问的，从而给与它最高的优先级。

在现有算法的局限性下，会导致缓存数据的命中率或多或少的受损，而命中略又是缓存的重要指标。HighScalability网站刊登了一篇文章，由前Google工程师发明的W-TinyLFU——一种现代的缓存 。Caffine Cache就是基于此算法而研发。Caffeine 因使用 Window TinyLfu 回收策略，提供了一个近乎最佳的命中率。

当数据的访问模式不随时间变化的时候，LFU的策略能够带来最佳的缓存命中率。然而LFU有两个缺点：

● 首先，它需要给每个记录项维护频率信息，每次访问都需要更新，这是个巨大的开销；

● 其次，如果数据访问模式随时间有变，LFU的频率信息无法随之变化，因此早先频繁访问的记录可能会占据缓存，而后期访问较多的记录则无法被命中。

因此，大多数的缓存设计都是基于LRU或者其变种来进行的。相比之下，LRU并不需要维护昂贵的缓存记录元信息，同时也能够反应随时间变化的数据访问模式。然而，在许多负载之下，LRU依然需要更多的空间才能做到跟LFU一致的缓存命中率。因此，一个“现代”的缓存，应当能够综合两者的长处。

TinyLFU维护了近期访问记录的频率信息，作为一个过滤器，当新记录来时，只有满足TinyLFU要求的记录才可以被插入缓存。如前所述，作为现代的缓存，它需要解决两个挑战：

● 一个是如何避免维护频率信息的高开销；

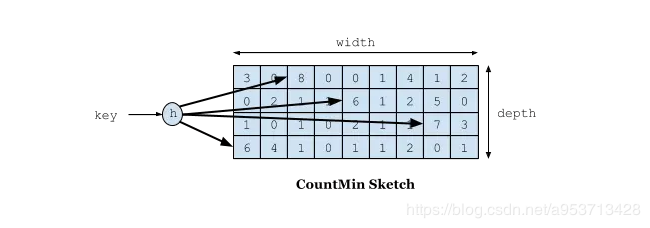
● 另一个是如何反应随时间变化的访问模式。

首先来看前者，TinyLFU借助了数据流Sketching技术，Count-Min Sketch显然是解决这个问题的有效手段，它可以用小得多的空间存放频率信息，而保证很低的False Positive Rate。但考虑到第二个问题，就要复杂许多了，因为我们知道，任何Sketching数据结构如果要反应时间变化都是一件困难的事情，在Bloom Filter方面，我们可以有Timing Bloom Filter，但对于CMSketch来说，如何做到Timing CMSketch就不那么容易了。TinyLFU采用了一种基于滑动窗口的时间衰减设计机制，借助于一种简易的reset操作：每次添加一条记录到Sketch的时候，都会给一个计数器上加1，当计数器达到一个尺寸W的时候，把所有记录的Sketch数值都除以2，该reset操作可以起到衰减的作用。

W-TinyLFU主要用来解决一些稀疏的突发访问元素。在一些数目很少但突发访问量很大的场景下，TinyLFU将无法保存这类元素，因为它们无法在给定时间内积累到足够高的频率。因此W-TinyLFU就是结合LFU和LRU，前者用来应对大多数场景，而LRU用来处理突发流量。

在处理频率记录的方案中，你可能会想到用hashMap去存储，每一个key对应一个频率值。那如果数据量特别大的时候，是不是这个hashMap也会特别大呢。由此可以联想到 Bloom Filter，对于每个key，用n个byte每个存储一个标志用来判断key是否在集合中。原理就是使用k个hash函数来将key散列成一个整数。

在W-TinyLFU中使用Count-Min Sketch记录我们的访问频率，而这个也是布隆过滤器的一种变种。如下图所示:



如果需要记录一个值，那我们需要通过多种Hash算法对其进行处理hash，然后在对应的hash算法的记录中+1，为什么需要多种hash算法呢？由于这是一个压缩算法必定会出现冲突，比如我们建立一个byte的数组，通过计算出每个数据的hash的位置。

比如张三和李四，他们两有可能hash值都是相同，比如都是1那byte[1]这个位置就会增加相应的频率，张三访问1万次，李四访问1次那byte[1]这个位置就是1万零1，如果取李四的访问评率的时候就会取出是1万零1，但是李四命名只访问了1次啊，为了解决这个问题，所以用了多个hash算法可以理解为long[][]二维数组的一个概念，比如在第一个算法张三和李四冲突了，但是在第二个，第三个中很大的概率不冲突，比如一个算法大概有1%的概率冲突，那四个算法一起冲突的概率是1%的四次方。通过这个模式我们取李四的访问率的时候取所有算法中，李四访问最低频率的次数。所以他的名字叫Count-Min Sketch。