```
            .maximumSize(10_000).buildAsync();
        String key =”test”;
        CompletableFuture<String> res = cache.get(key,k-> getValue(key));
        res.thenAccept(result -> System.out.println(result));
    }

    // 模拟从外部数据源加载数据的逻辑
    private static String getValue(String key) {
        return ”value”;
    }
```

```

异步加载使用的类是AsyncCache，使用方法和Cache类似。cache.get(key, k -> getValue(key))将会返回一个CompletableFuture，这一步骤会在一个异步任务中执行，而不会阻塞主线程。res.thenAccept方法将在数据加载完成后得到结果。

### 1.4自动异步加载:

```
    public static void main(String[] args) throws Exception {
        Executor executor = Executors.newFixedThreadPool(5);
        AsyncLoadingCache<String, String> cache = Caffeine.newBuilder()
                .expireAfterWrite(10, TimeUnit.MINUTES)
                .maximumSize(10_000)
                //异步的封装一段同步操作来生成缓存元素
                .buildAsync(key -> getValue(key))
                //OR建一个异步缓存元素操作并返回一个future
                .buildAsync((key,executor1) -> getValue(key,executor));
        String key = ”test”;
        CompletableFuture<String> res = cache.get(key);
        res.thenAccept(result -> System.out.println(result));
    }

    // 模拟从外部数据源加载数据的逻辑
    private static CompletableFuture<String> getValue(String
key,Executor executor) {
        return CompletableFuture.supplyAsync(() -> ”value for ” + key,
executor);
```
```

```
    }
     private static String getValue(String key) {
         return ”value”;
    }


```

自动异步加载使用方法和手动异步加载类似，getValue可接收一个Executor对象，用于自定义执行异步操作的线程池。

2.驱逐：

2.1基于容量：

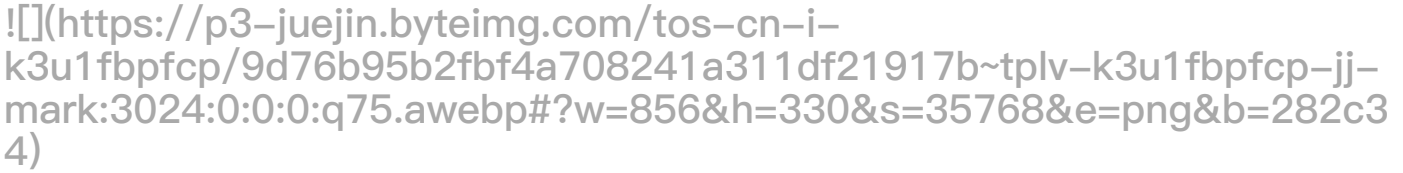
```
Cache<String, String> cache = Caffeine.newBuilder()
                .maximumSize(10_000)
                .build();
```

最常用的驱逐策略了，Caffeine提供多种算法根据最近使用频率和使用时间来驱逐元素 ref：[Window-TinyLFU](http://cxyroad.com/”https://github.com/ben-manes/caffeine/wiki/Efficiency-zh-CN”)

2.2基于权重：

```
class Product {
    private String name;
    private int weight;

    public Product(String s, int i) {
        name=s;
        weight=i;
    }
    public String getName() {
        return name;
```

```
    }
    public int getWeight() {
        return weight;
    }
    @Override
    public String toString(){
        return getName();
    }
}

public class TestCaffeineCache {
    public static void main(String[] args) {
        Cache<String, Product> cache = Caffeine.newBuilder()
                .maximumWeight(1000)
                .weigher((String key, Product value) -> value.getWeight())
                //使用当前线程进行驱逐和刷新
                .executor(runnable -> runnable.run())
                //监听器，如果有元素被驱逐则会输出
                .removalListener(((key, value, cause) -> {
                    System.out.printf("Key %s was evicted (%s)%n", key,
cause);
                }))
                .build();
        // 向缓存中添加商品信息
        cache.put("product1", new Product("Product 1", 200));
        cache.put("product2", new Product("Product 2", 400));
        cache.put("product3", new Product("Product 3", 500));
        // 获取缓存中的商品信息
        System.out.println(cache.getIfPresent("product1"));
        System.out.println(cache.getIfPresent("product2"));
        System.out.println(cache.getIfPresent("product3"));
    }
}

```



.weigher((String key, Product value) -> value.getWeight()) 制定了一个权重计算器，Product对象的getWeight()方法来计算权重。

通过示例中的返回结果可以看到，当product3被put后，总容量超过了1000，product1就被驱逐了。

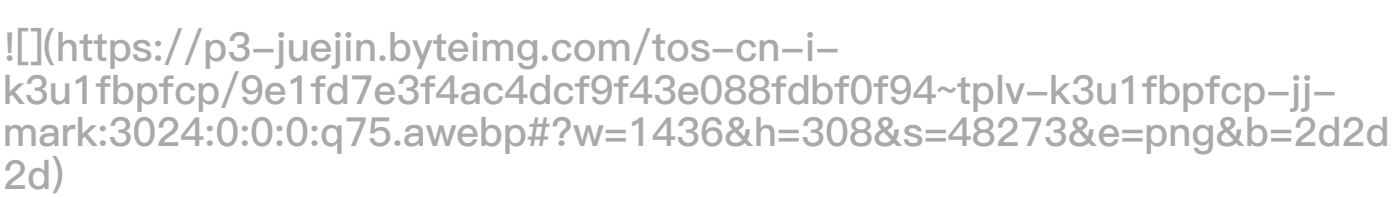
2.3基于时间：

附上官方的例子：

```
// 基于固定的过期时间驱逐策略
LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()
    .expireAfterAccess(5, TimeUnit.MINUTES)
    .build(key -> createExpensiveGraph(key));
LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()
    .expireAfterWrite(10, TimeUnit.MINUTES)
    .build(key -> createExpensiveGraph(key));

// 基于不同的过期驱逐策略
LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()
    .expireAfter(new Expiry<Key, Graph>() {
      public long expireAfterCreate(Key key, Graph graph, long currentTime) {
        // Use wall clock time, rather than nanotime, if from an external resource
        long seconds = graph.creationDate().plusHours(5)
            .minus(System.currentTimeMillis(), MILLIS)
            .toEpochSecond();
        return TimeUnit.SECONDS.toNanos(seconds);
      }
      public long expireAfterUpdate(Key key, Graph graph,
          long currentTime, long currentDuration) {
        return currentDuration;
      }
      public long expireAfterRead(Key key, Graph graph,
          long currentTime, long currentDuration) {
        return currentDuration;
      }
    })
    .build(key -> createExpensiveGraph(key));
```

```
Caf          @Override
                 public String reload(String s,String v){
                     return getVale(s)+"|"+v;
                 }
             }); // 提供加载方法
        System.out.println("Initial value for key1: " + cache.get("key1"));
        // 等待超过自动刷新时间
        ticker.advance(7, TimeUnit.SECONDS);
        cache.get("key1");
        System.out.println(cache.get("key1")); // 输出自动刷新后的值
    }

    private static String getVale(String key) {
        // 这里简单地返回一个当前时间的字符串
        return "loaded value for " + key + " at " +
System.currentTimeMillis();
    }

```

结果：



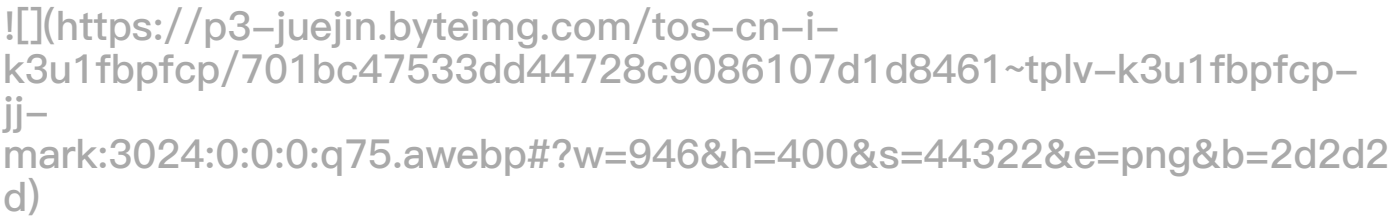
三、性能对比：
=======

我们知道，Caffeine的性能比Guava Cache要好，可以写一个demo简单对比一下：

1.Caffeine Demo:

```
package test;

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

public class CaffeineCacheTest {

    public static void main(String[] args) throws Exception {
        Cache<Integer, Integer> loadingCache = Caffeine.newBuilder()
                .build();

        // 开始时间
        Long start = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.put(i, i);
        }
        // 存完成时间
        Long writeFinishTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.getIfPresent(i);
        }
        // 读取完成时间
        Long readFinishTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.invalidate(i);
        }
        // 删除完成时间
        Long deleteFinishTime = System.currentTimeMillis();
        System.out.println("CaffeineCache存用时：" + (writeFinishTime - start));
        System.out.println("CaffeineCache读用时：" + (readFinishTime - writeFinishTime));
        System.out.println("CaffeineCache删用时：" + (deleteFinishTime - readFinishTime));
    }
}
```

运行结果：



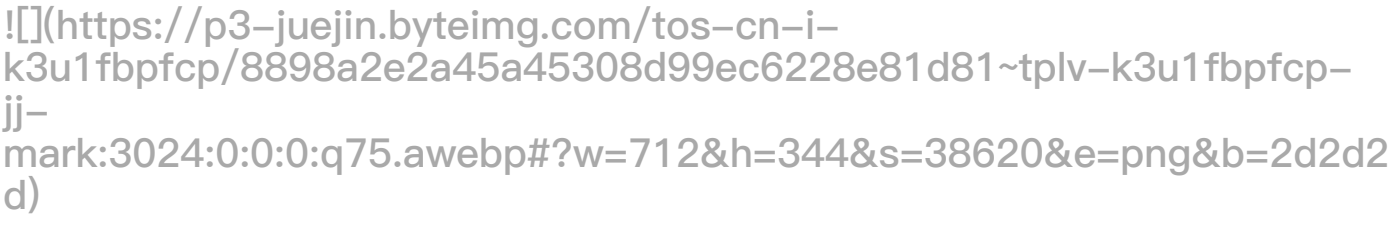
2.Guava Cache Demo:

使用几乎一致的API，换成Guava Cache再试一次：

```
package test;

import com.google.common.cache.Cache;
import com.google.common.cache.CacheBuilder;

public class GuavaCacheTest {

    public static void main(String[] args) throws Exception {
        Cache<Integer, Integer> loadingCache = CacheBuilder.newBuilder()
                .build();

        // 开始时间
        Long start = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.put(i, i);
        }
        // 存完成时间
        Long writeFinishTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.getIfPresent(i);
        }
        // 读取完成时间
        Long readFinishTime = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            loadingCache.invalidate(i);
        }
```

```
        // 删除完成时间
        Long deleteFinishTime = System.currentTimeMillis();
        System.out.println("GuavaCache存用时："+(writeFinishTime-start));
        System.out.println("GuavaCache取用时："+(readFinishTime-
writeFinishTime));
        System.out.println("GuavaCache删用时："+(deleteFinishTime-
readFinishTime));
    }

}
```

运行结果：



3.多组测试结果：

运行环境：处理器：Apple M3 ，内存：18 GB，JDK1.8

更改循环次数，多组测试结果如下（单位ms）：

缓存	Caffeine	Guava Cache	Caffeine	Guava Cache	Caffeine	Guava Cache	Caffeine	Guava Cache
次数	100	100	10000	10000	1000000	1000000	5000000	5000000
存用时	1	2	17	51	113	279	3802	2458
取用时	0	0	9	16	25	141	47	531

| 删用时 | 0 | 11 | 7 | 25 | 35 | 176 | 89 | 1073 |

可以看出Caffeine的总体性能是比Guava Cache要好的。

当然，基于本地单机的简单测试，结果受处理器，线程，内存等影响较大。可以参考下官方的测试，有更高的参考意义：[官方测试。](http://cxyroad.com/”https://github.com/ben-manes/caffeine/wiki/Benchmarks-zh-CN”)

四、总结：
=====

本文举了很多的例子，介绍了Caffeine支持的多种基础的操作，包括存、取、删等。以及异步、监听、刷新等更多拓展的操作，能够覆盖大部分需要本地缓存的开发场景。

Caffeine的性能比Guava Cache更好，并列举了一个性能测试 demo，Caffeine兼容Guava Cache的API，所以从Guava Cache迁移至Caffeine也比较容易。

最后附上Caffeine的官方网址：[官方网址（中文）。](http://cxyroad.com/”https://github.com/ben-manes/caffeine/wiki/Home-zh-CN”)

作者：京东物流 殷世杰

来源：京东云开发者社区

原文链接: https://juejin.cn/post/7369535515291500578