**缓存背景**

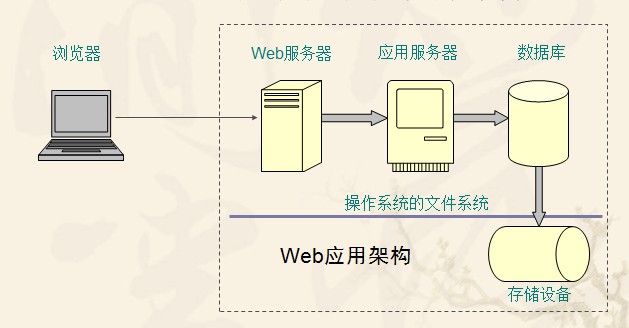
缓存是我们在开发中为了提高系统的性能，把经常的访问业务的数据第一次把处理结果先放到缓存中，第二次就不用在对相同的业务数据在重新处理一遍，这样可以减少系统开销，提高了系统的性能。

缓存主要可分为二大类:

一、文件缓存: 顾名思义文件缓存是指把数据存储在磁盘上，不管你是以XML格式，序列化文件DAT格式还是其它文件格式；

二、内存缓存: 将缓存数据存储在服务器内存当中，例如实现一个类中静态Map,对这个Map进行常规的增删查.

其中又包括了几种实现方式：1,本地缓存   2,数据库缓存    3,分布式缓存



eg：OSCache ， Guava Cache，Redis  , EHCache，Memcached ......

参考资料：

<http://blog.csdn.net/baidu_33497625/article/details/50801571>

<http://blog.csdn.net/xiaodingdou/article/details/52562926>

**适用场景**

**缓存在很多场景下都是相当有用的。例如，计算或检索一个值的代价很高，同时需要频繁获取该值的时候，就应当考虑使用缓存。**

**内存缓存：**

* 你愿意消耗一部分内存来提升速度；
* 你已经预料某些值会被多次调用；
* 缓存数据不会超过内存总量；

引申：MIGU项目中，我们使用到的Redis和本地缓存之间的关系。

**一，ConcurrentHashMap**

**线程不安全的HashMap**

常用：Map map = new HashMap();

    HashMap不是线程安全的，主要是并发多线程同时put时，如果同时触发了rehash操作，会导致HashMap中的链表中出现循环节点，进而使得后续get出现死循环。

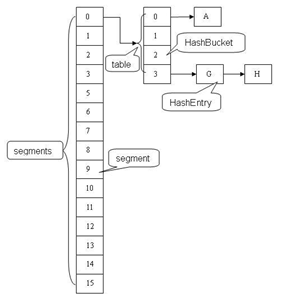
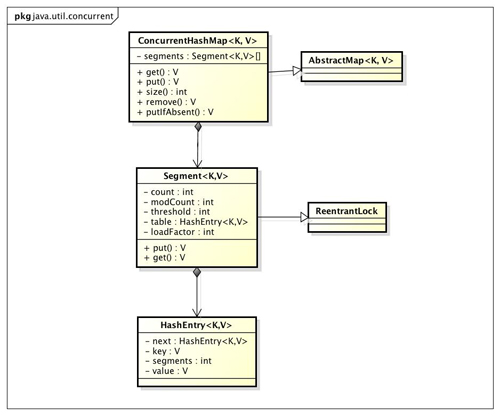
(rehash：HashMap初始容量大小为16,一般来说，当有数据要插入时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的元素都需要被重算一遍。)

**效率低下的HashTable容器**

    HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，竞争越激烈效率越低。

**锁分段技术**

    HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种**可重入锁ReentrantLock**，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

ConcurrentHashMap（JDK1.5新增concurrent包）是一个线程安全的HashTable，并提供了一组和HashTable功能相同但是线程安全的方法。

由于ConcurrentMap是一个接口类，使用时需要使用它的实现类。Java.util.concurrent包中有下面关于ConcurrentMap接口的实现类。

ConcurrentHashMap读取数据的时候不加锁，提高了并发能力。

ConcurrenthashMap在写的时候不会全部加锁，仅仅锁住Map中正在被写入的部分。

ConcurrentMap concurrentMap = new ConcurrentHashMap();

concurrentMap.put("key", "value");

Object value = concurrentMap.get("key");

concurrentMap.remove("key");

concurrentMap.clear();

或者

ConcurrentMap<String,Book> books = new MapMaker().concurrencyLevel(2) .softValues() .makeMap();

**ConcurrentHashMap优势特点：**

* ConcurrentHashMap的所有操作都是线程安全
* ConcurrentHashMap允许并发的读和线程安全的更新操作
* 在执行写操作时，ConcurrentHashMap只锁住部分的Map
* 并发的更新是通过内部根据并发级别将Map分割成小部分实现的
* 可以使用ConcurrentHashMap代替HashTable，但ConcurrentHashMap不会锁住整个Map

**ConcurrentHashMap问题：**

1，并发场景问题？

2，内存元素回收  或者  在保存数据会出现内存溢出如何处理？

查看Test实例：用ConcurrentHashMap模拟本地缓存，当在高并发环境一下，会出现一些什么问题？

问题结论：

高并发访问缓存支持不好

没有资源回收释放机制，容易造成内存溢出，Google Guava Cache在这些问题方面都做得挺好的。

Guava Cache与ConcurrentMap很相似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。

相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管LoadingCache 不回收元素，它也会自动加载缓存。

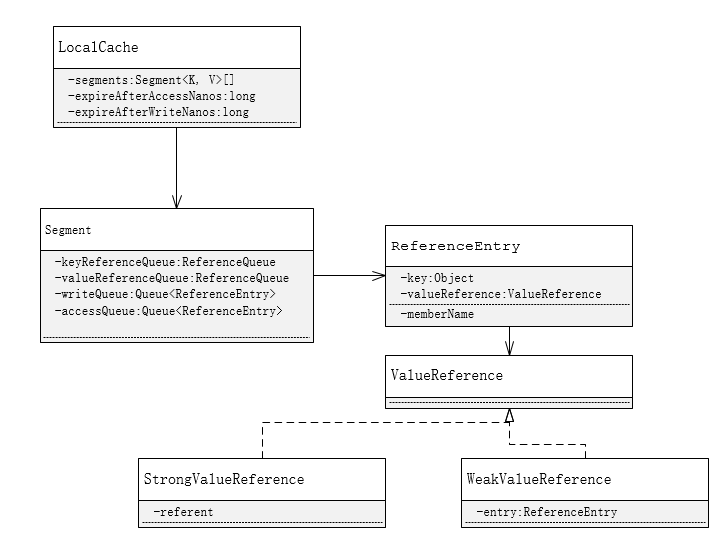
**二，Google Guava**

Guava工程包含了若干被Google的Java项目广泛依赖的开源核心库，例如：集合 [collections] 、缓存 [caching] 、原生类型支持 [primitives support] 、并发库 [concurrency libraries] 、通用注解 [common annotations] 、字符串处理 [string processing] 、I/O 等等

**缓存 [caching]**

Guava通过接口LoadingCache提供了一个非常强大的基于内存的LoadingCache<K，V>。

在缓存中自动加载值，它提供了许多实用的方法，在有缓存需求时非常有用。



基本的接口 Cache 和LoadingCache，其中LoadingCache接口继承了Cache接口

**范例**

LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()

.maximumSize(1000)

.expireAfterWrite(10, TimeUnit.MINUTES)

.removalListener(MY\_LISTENER)

.build(

    new CacheLoader<Key, Graph>() {

        public Graph load(Key key) throws AnyException {

            return createExpensiveGraph(key);

    }

});

Cache实例通过CacheBuilder生成器模式获取

**LoadingCache的设计是线程安全**的，针对同一个key的并发调用是会被block的，

一旦value获取到了，这个value值就会返回给调用的get方法，但是如果调用get是多个不同的key那么我们就可以进行并发调用。

**缓存刷新**

刷新和回收不太一样。正如[LoadingCache.refresh(K)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/LoadingCache.html#refresh%28K%29)所声明，**刷新表示为键加载新值，这个过程可以是异步的**。在刷新操作进行时，缓存仍然可以向其他线程返回旧值，而不像回收操作，读缓存的线程必须等待新值加载完成。

如果刷新过程抛出异常，缓存将保留旧值，而异常会在记录到日志后被丢弃。

[CacheBuilder.refreshAfterWrite(long, TimeUnit)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#refreshAfterWrite%28long,%20java.util.concurrent.TimeUnit%29)可以为缓存增加自动定时刷新功能。和expireAfterWrite相反，refreshAfterWrite通过定时刷新可以让缓存项保持可用，但请注意：缓存项只有在被检索时才会真正刷新（如果CacheLoader.refresh实现为异步，那么检索不会被刷新拖慢）。因此，如果你在缓存上同时声明expireAfterWrite和refreshAfterWrite，缓存并不会因为刷新盲目地定时重置，如果缓存项没有被检索，那刷新就不会真的发生，缓存项在过期时间后也变得可以回收。

**缓存回收**

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

**1，基于容量的回收（size-based eviction）**

如果要规定缓存项的数目不超过固定值，只需使用[CacheBuilder.maximumSize(long)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#maximumSize%28long%29)。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。

另外，不同的缓存项有不同的“权重”（weights）——例如，如果你的缓存值，占据完全不同的内存空间，你可以使用[CacheBuilder.weigher(Weigher)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#weigher%28com.google.common.cache.Weigher%29)指定一个权重函数，并且用[CacheBuilder.maximumWeight(long)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#maximumWeight%28long%29)指定最大总重。

**2，定时回收（Timed Eviction）**

CacheBuilder提供两种定时回收的方法：

[expireAfterAccess(long, TimeUnit)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#expireAfterAccess%28long,%20java.util.concurrent.TimeUnit%29)：缓存项在给定时间内没有被读/写访问，则回收。

[expireAfterWrite(long, TimeUnit)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/cache/CacheBuilder.html#expireAfterWrite%28long,%20java.util.concurrent.TimeUnit%29)：缓存项在给定时间内没有被写访问（创建或覆盖），则回收。

**3，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（见代码）

**显式清除**

任何时候，你都可以显式地清除缓存项，而不是等到它被回收：

* 个别清除：[Cache.invalidate(key)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/Cache.html#invalidate%28java.lang.Object%29)
* 批量清除：[Cache.invalidateAll(keys)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/Cache.html#invalidateAll%28java.lang.Iterable%29)
* 清除所有缓存项：[Cache.invalidateAll()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/Cache.html#invalidateAll%28%29)

**清理什么时候发生？**

使用CacheBuilder构建的缓存不会"自动"执行清理和回收工作，也不会在某个缓存项过期后马上清理，也没有诸如此类的清理机制。相反，它会**在写操作时顺带做少量的维护工作，或者偶尔在读操作时做**——如果写操作实在太少的话。

这样做的原因在于：如果要自动地持续清理缓存，就必须有一个线程，这个线程会和用户操作竞争共享锁。此外，某些环境下线程创建可能受限制，这样CacheBuilder就不可用了。

**缓存命中统计**

[CacheBuilder.recordStats()](http://docs.guava-libraries.googlecode.com/git-history/release12/javadoc/com/google/common/cache/CacheBuilder.html#recordStats%28%29)用来开启Guava Cache的统计功能。统计打开后，[Cache.stats()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/Cache.html#stats%28%29)方法会返回[CacheStats](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheStats.html)对象以提供如下统计信息：

* [hitRate()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheStats.html#hitRate%28%29)：缓存命中率；
* [averageLoadPenalty()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheStats.html#averageLoadPenalty%28%29)：加载新值的平均时间，单位为纳秒；
* [evictionCount()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheStats.html#evictionCount%28%29)：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**。

**Google Guava做内存缓存**

google guava中有cache包，此包提供内存缓存功能。内存缓存需要考虑很多问题，包括并发问题，缓存失效机制，内存不够用时缓存释放，缓存的命中率，缓存的移除等等。当然这些东西guava都考虑到了。

guava的内存缓存非常强大，可以设置各种选项，而且很轻量，使用方便。另外还提供了下面一些方法，来方便各种需要：

put和putAll方法向缓存中添加一个或者多个缓存项

invalidate 和 invalidateAll方法从缓存中移除缓存项

asMap()方法获得缓存数据的ConcurrentMap<K, V>快照

cleanUp()清空缓存

refresh(Key) 刷新缓存，即重新取缓存数据，更新缓存

**Google Guava缓存过期特性：**

guava缓存过期时间分为两种，一种是从写入时开始计时（expireAfterWrite），一种是从最后访问时间开始计时（expireAfterAccess），而且guava缓存的过期时间是设置到整个一组缓存上的；这和EHCache，redis，memcached等不同，这些缓存系统设置都将缓存时间设置到了单个缓存上。

guava缓存设计成了一组对象一个缓存实例，这样做的好处是一组对象设置一组缓存策略，你可以根据不同的业务来设置不同的缓存策略，包括弱引用，软引用，过期时间，最大项数等。另外一点好处是你可以根据不同的组来统计缓存的命中率，这样更有意义一些。

**扩展：**

**put操作**

put操作相对比较简单，首先它需要获得锁，然后尝试做一些清理工作，接下来的逻辑类似ConcurrentHashMap中的rehash，不详述。需要说明的是当找到一个已存在的Entry时，需要先判断当前的ValueRefernece中的值事实上已经被回收了，因为它们可以时WeakReference、SoftReference类型，如果已经被回收了，则将新值写入。并且在每次更新时注册当前操作引起的移除事件，指定相应的原因：COLLECTED、REPLACED等，这些注册的事件在退出的时候统一调用LocalCache注册的RemovalListener，由于事件处理可能会有很长时间，因而这里将事件处理的逻辑在退出锁以后才做。最后，在更新已存在的Entry结束后都尝试着将那些已经expire的Entry移除。另外put操作中还需要更新writeQueue和accessQueue的语义正确性。

**get操作**

大概上的步骤是：1. 先查找table中是否已存在没有被回收、也没有expire的entry，如果找到，并在CacheBuilder中配置了refreshAfterWrite，并且当前时间间隔已经超过这个时间，则重新加载值，否则，直接返回原有的值；2. 如果查找到的ValueReference是LoadingValueReference，则等待该LoadingValueReference加载结束，并返回加载的值；3. 如果没有找到entry，或者找到的entry的值为null，则加锁后，继续table中已存在key对应的entry，如果找到并且对应的entry.isLoading()为true，则表示有另一个线程正在加载，因而等待那个线程加载完成，如果找到一个非null值，返回该值，否则创建一个LoadingValueReference，并调用loadSync加载相应的值，在加载完成后，将新加载的值更新到table中，即大部分情况下替换原来的LoadingValueReference。

参考资料：

Grava教程：<http://ifeve.com/google-guava/>

Grava缓存：<http://ifeve.com/google-guava-cachesexplained/>

Grava Jar类库（Maven）：<https://mvnrepository.com/artifact/com.google.guava/guava>