性能优化：不得不说的缓存技巧

利用缓存做性能优化的案例非常多，从基础的操作系统到数据库、分布式缓存、本地缓存等。它们表现形式各异，却有着共同的朴素的本质：弥补CPU的高算力和IO的慢读写之间巨大的鸿沟。

和架构选型类似，每引入一个组件，都会导致复杂度的上升。以缓存为例，它带来性能提升的同时，也带来一些问题，需要开发者设计和权衡。

思维脉络如下：

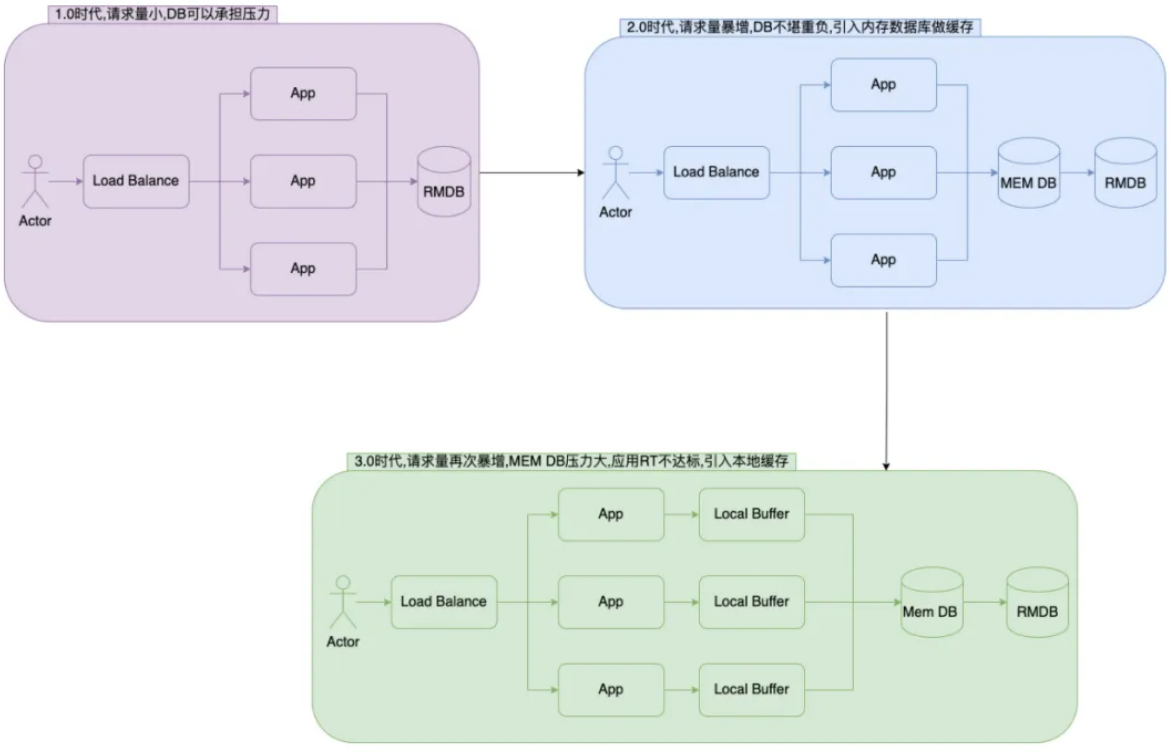
一 缓存和多级缓存

1 缓存的引入

在初期业务量小的时候，数据库能承担读写压力，应用可以直接和DB交互，架构简单且强壮。

经过一段时间发展后，业务量迎来了大规模增长，此时DB查询压力和耗时都在增长。此时引入分布式缓存，在减少DB压力的同时，还提供了更高的QPS。

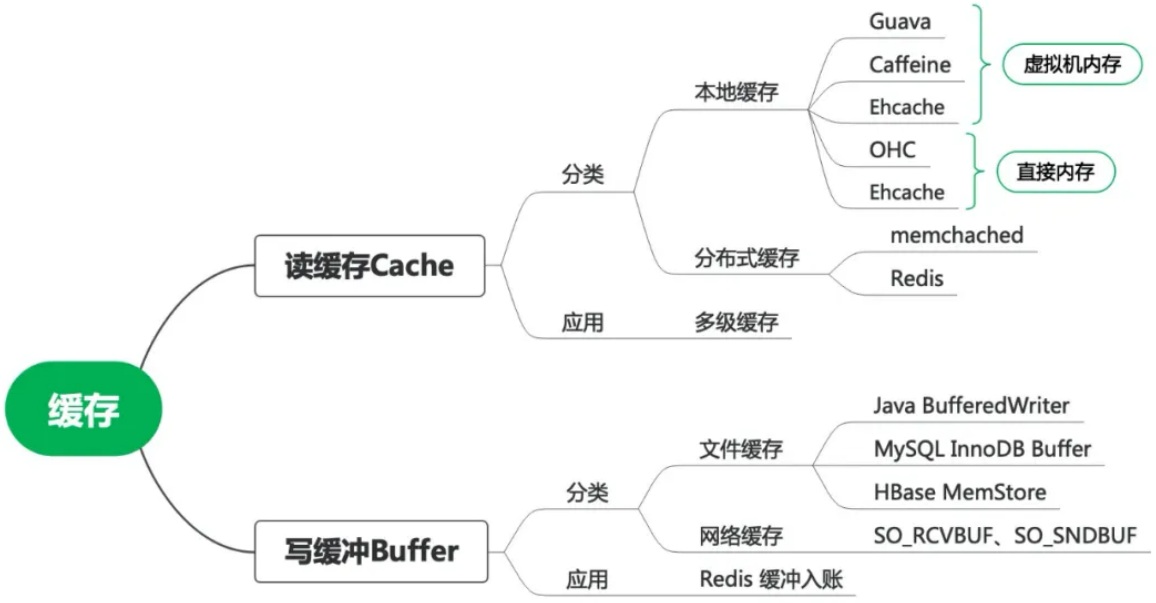
再往后发展，分布式缓存也成为了瓶颈，高频的QPS是一笔负担;另外缓存驱逐以及网络抖动会影响系统的稳定性，此时引入本地缓存，可以减轻分布式缓存的压力，并减少网络以及序列化开销。



2 读写的性能提升

缓存通过减少IO操作来获得读写的性能提升。有一个表格，可以看见磁盘、网络的IO操作耗时，远高于内存存取。

- 读优化：当请求命中缓存后，可直接返回，从而略过IO读取，减小读的成本。
- 写优化：将写操作在缓冲中合并，让IO设备可以批量处理，减小写的成本。



缓存带来的QPS、RT提升比较直观，不补充介绍。

3 缓存Miss

缓存Miss是必然会面对的问题，缓存需保证在有限的容量下，将热点的数据维护在缓存中，从而达到性能、成本的平衡。

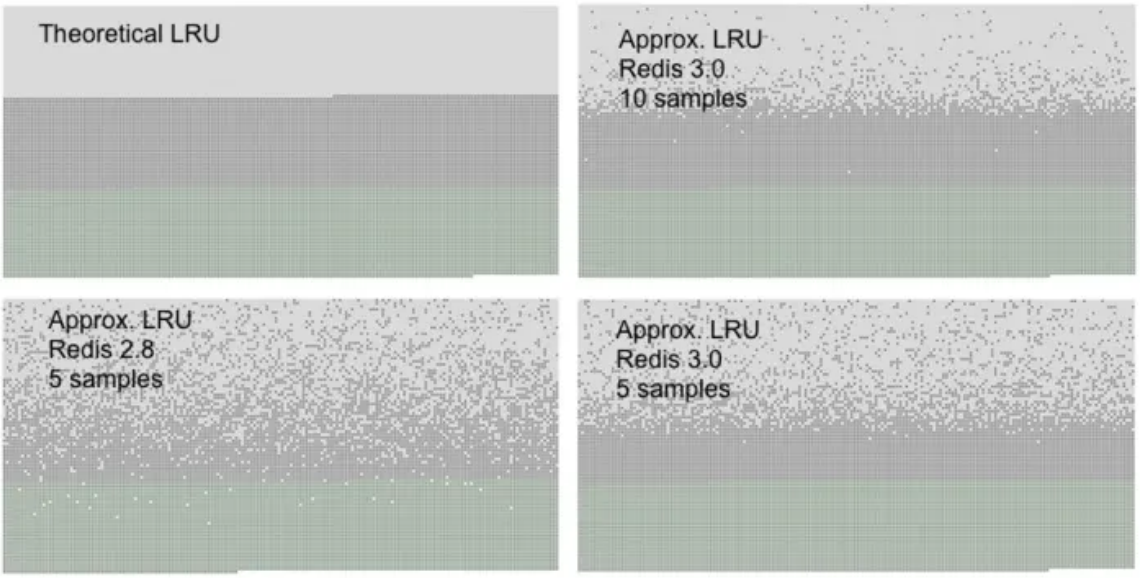
缓存通常使用LRU算法淘汰近期不常用的Key。

近似LRU

可以先试想严格LRU的实现。假设Redis当前有50W规模的key，先通过Keys 遍历获得所有Key，然后比对出空闲时间最长的某个key，最后执行淘汰。这样的流程下来，是非常昂贵的，Keys命令是一笔不小的开销，其次大规模执行比对也很昂贵。

当然严格LRU实现的优化空间还是有的，YY一下，可以通过活跃度分离出活跃Key和待回收Key， 淘汰时只关注待回收key即可;回收算法引入链表或者树的结构，使Key按空闲时间有序，淘汰时直接获取。然而这些优化不可避免的是，在缓存读写时，这些辅助的数据结构需要同步更新，带来的存储以及计算的成本很高。

在Redis中它采用了近似LRU的实现，它随机采样5个Key，淘汰掉其中空闲时间最长的那个。近似LRU实现起来更简单、成本更低，在效果上接近严格LRU。它的缺点是存在一定的几率淘汰掉最近被访问的Key，即在TTL到期前也可能被淘汰。



避免短期大量失效

在一些场景中，程序是批量加载数据到缓存的， 比如通过Excel上传数据，系统解析后，批量写入DB和缓存。此时若不经设计，这批数据的超时时间往往是一致的。缓存到期后，本该缓存承担的流量将打到DB上，从而降低接口甚至系统的性能和稳定性。

可以利用随机数打散缓存失效时间，例如设置TTL=8hr+random(8000)ms。

4 缓存一致性

系统应尽量保证DB、缓存的数据一致性，较常使用的是cache aside设计模式。

避免使用非常规的缓存设计模式：先更新缓存、后更新DB；先更新DB、后更新缓存(cache aside是直接失效缓存)。这些模式的不一致风险较高。

缓存设计模式

业务系统通常使用cache aside 模式，操作系统、数据库、分布式缓存等会使用write throgh、write back。

策略名词	文字解释	优缺点	典型案例
cache aside	先更新存储，成功后失效缓存。		业务系统
write through	同时写缓存和存储。	优点：每次写入存储，数据丢失风险低；缺点：存储交互较多，性能较差。	Redis apendfsync配置everysec，每间隔一秒中，调用一次fsync，将缓存写入到存储。
write back	只更新缓存，由缓存主导同步时机。	优点：可自主选择存储写入时机，性能较好；缺点：存在断电数据丢失风险。	例如Redis apendfsync配置no，由操作系统的cache与存储同步。

cache aside的缓存不一致

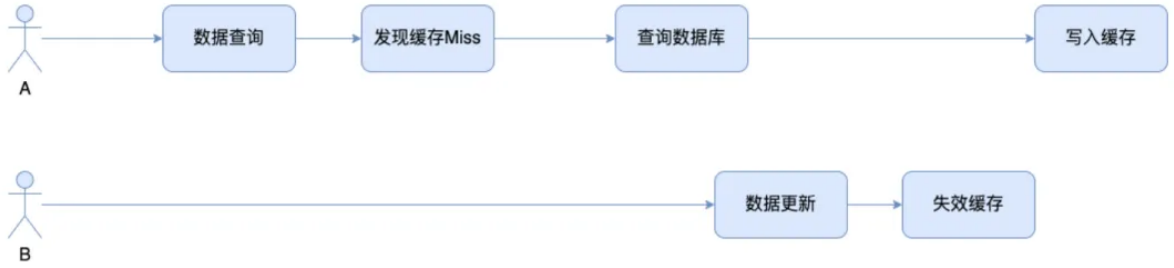
Cache aside模式大部分时间运行良好，在一些极端场景下，仍可能出现不一致风险。主要来自两方面：

1. 由于中间件或者网络等问题，缓存失效失败。

2. 出现意外的缓存失效、读取的时序。

缓存失效失败很容易理解，不做补充。主要介绍时序引起的不一致问题。

考虑这样的时间轴，A线程发现cache miss后重新加载缓存，此时读的数据还是老的， 另一个线程B更新数据并失效缓存。若B线程失效缓存的操作完成时间早于A线程，A线程会写入老的数据。



缓存不一致有一些缓解方法，例如延迟双删、CDC同步。这些方案都提升了系统复杂度，需综合考虑业务的容忍度，方案的复杂度等。

- 延迟双删：主线程失效缓存后，将失效指令放入延时队列，另一个线程轮询队列获取指令并执行。
- CDC同步：通过canal订阅MySQL binlog的变更，上报给Kafka，系统监听Kafka消息触发缓存失效。

二 从堆内存到直接内存

1 直接内存的引入

Java本地缓存分两类，基于堆内存的、基于直接内存的。

采用堆内存做缓存的主要问题是GC，由于缓存对象的生命周期往往较长，需要通过Major GC进行回收。若缓存的规模很大，那么GC会非常耗时。

采用直接内存做缓存的主要问题是内存管理。程序需自主控制内存的分配和回收，存在OOM或者Memory Leak的风险。另外直接内存不能存取对象，在操作时需进行序列化。

直接内存能减少GC压力，因为它只需要保存直接内存的引用，而对象本身是存储在直接内存中。引用晋升到老年代后占用的空间很小，对GC的负担可忽略。

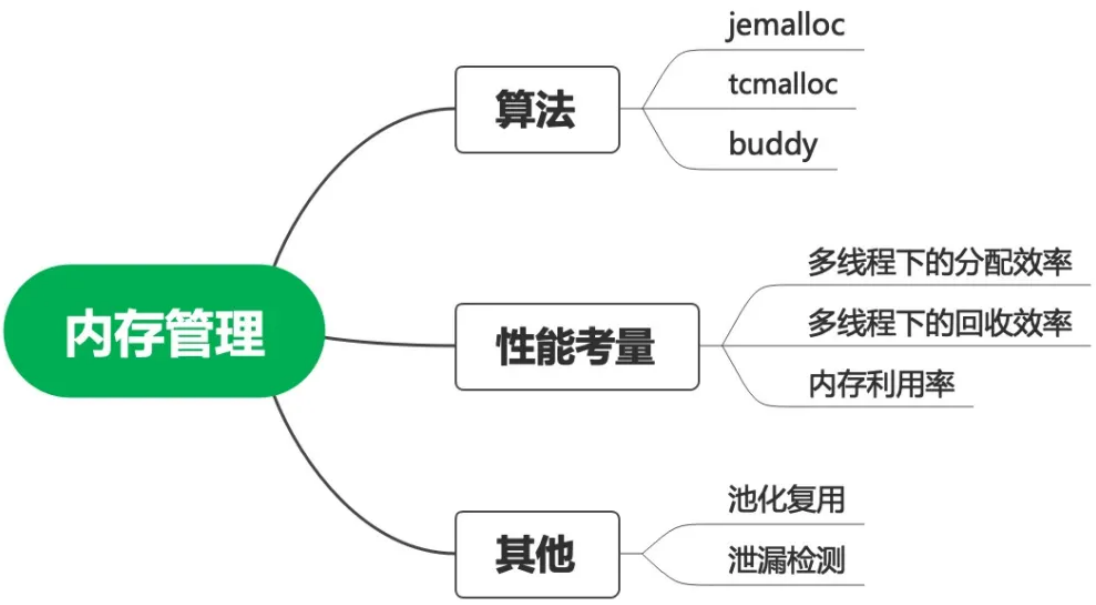
直接内存的回收依赖System。gc的调用，但这个调用JVM不保证执行、也不保证何时执行，它的行为是不可控的。程序一般需要自行管理，成对去调用malloc、free，依托于这种“手工、类C”的内存管理，可以增加内存回收的可控性和灵活性。

2 直接内存管理

由于直接内存的分配和回收比较昂贵，需要通过内核操作物理内存。申请的时候一般是申请大的内存快，然后再根据需求分配小块给线程。回收的时候不直接释放，而是放入内存池来重用。

如何快速找到一个空闲块、如何减少内存碎片、如何快速回收等等，它是一个系统性的问题，也有很多专门的算法。

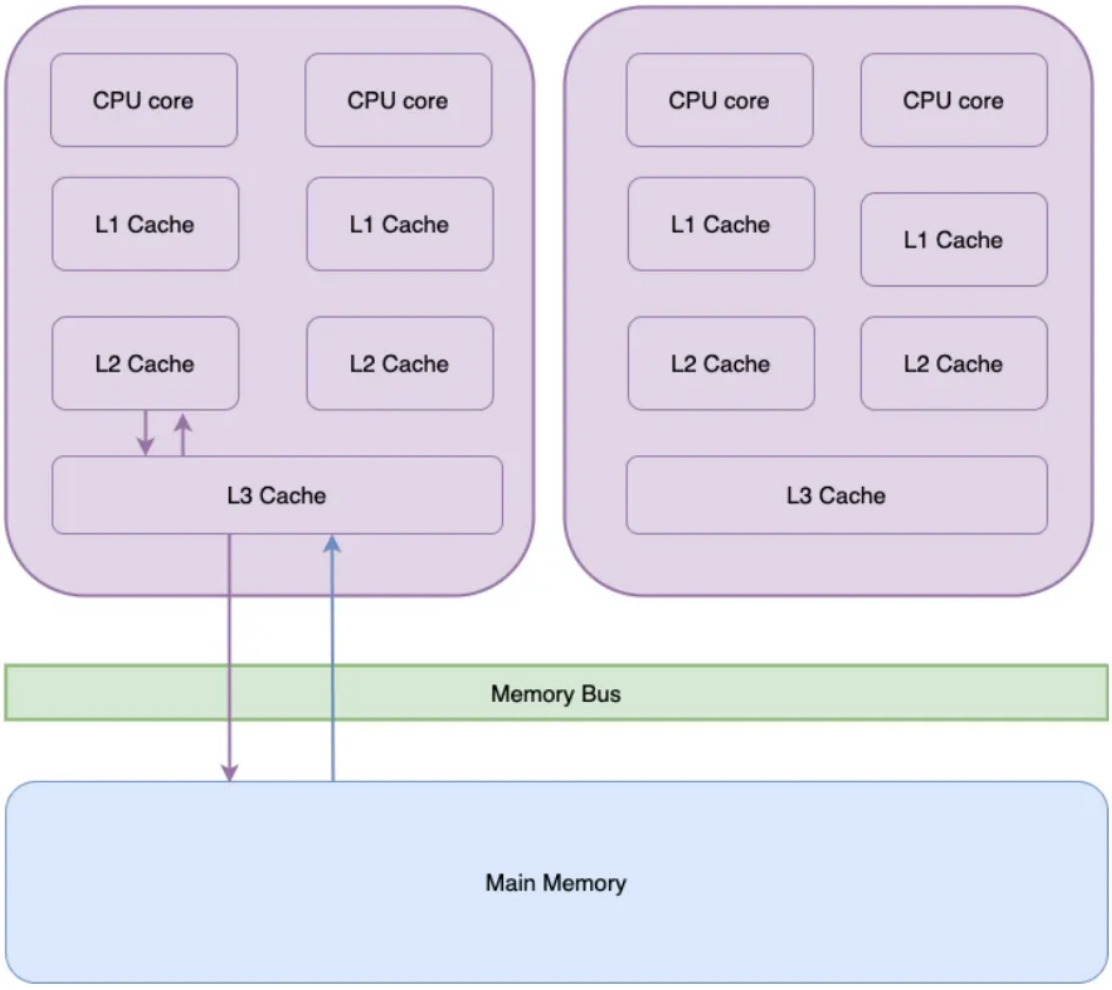
Jemalloc是综合能力较好的算法，free BSD、Redis默认采用了该算法，OHC缓存也建议服务器配置该算法。Netty的作者实现了Java版本，感兴趣的可以阅读。



三 CPU缓存

利用上分布式缓存、本地缓存之后，还可以继续提升的就是CPU缓存了。它虽不易察觉，但在高并发下对性能存在一定的影响。

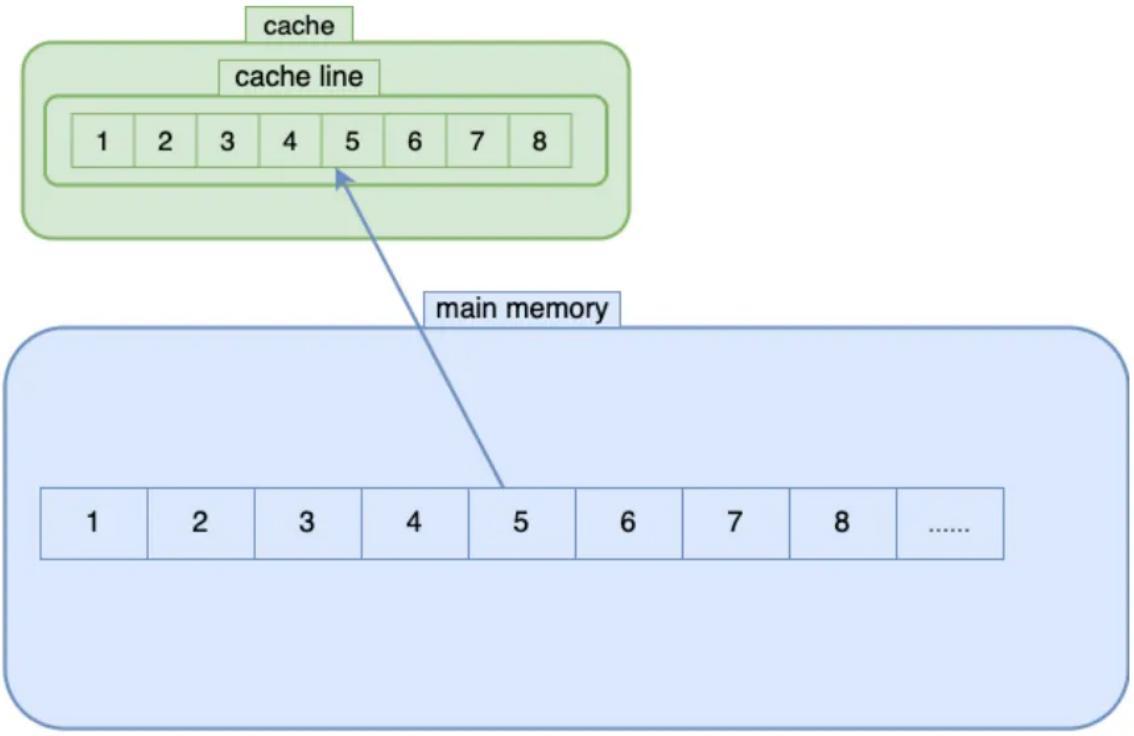
CPU缓存分为L1、L2、L3 三级，越靠近CPU的，容量越小，命中率越高。当L3等级的缓存都取不到数据的时候，需从主存中获取。



区域	CPU时钟周期	消耗时间
主存	-	约60-80ns
QPI 总线传输(between sockets， not drawn)	-	约20ns
L3 cache	约40-45 cycles	约15ns
L2 cache	约10 cycles	约3ns
L1 cache	约3-4 cycles	约1ns
寄存器	1 cycle	-

1 CPU cache line

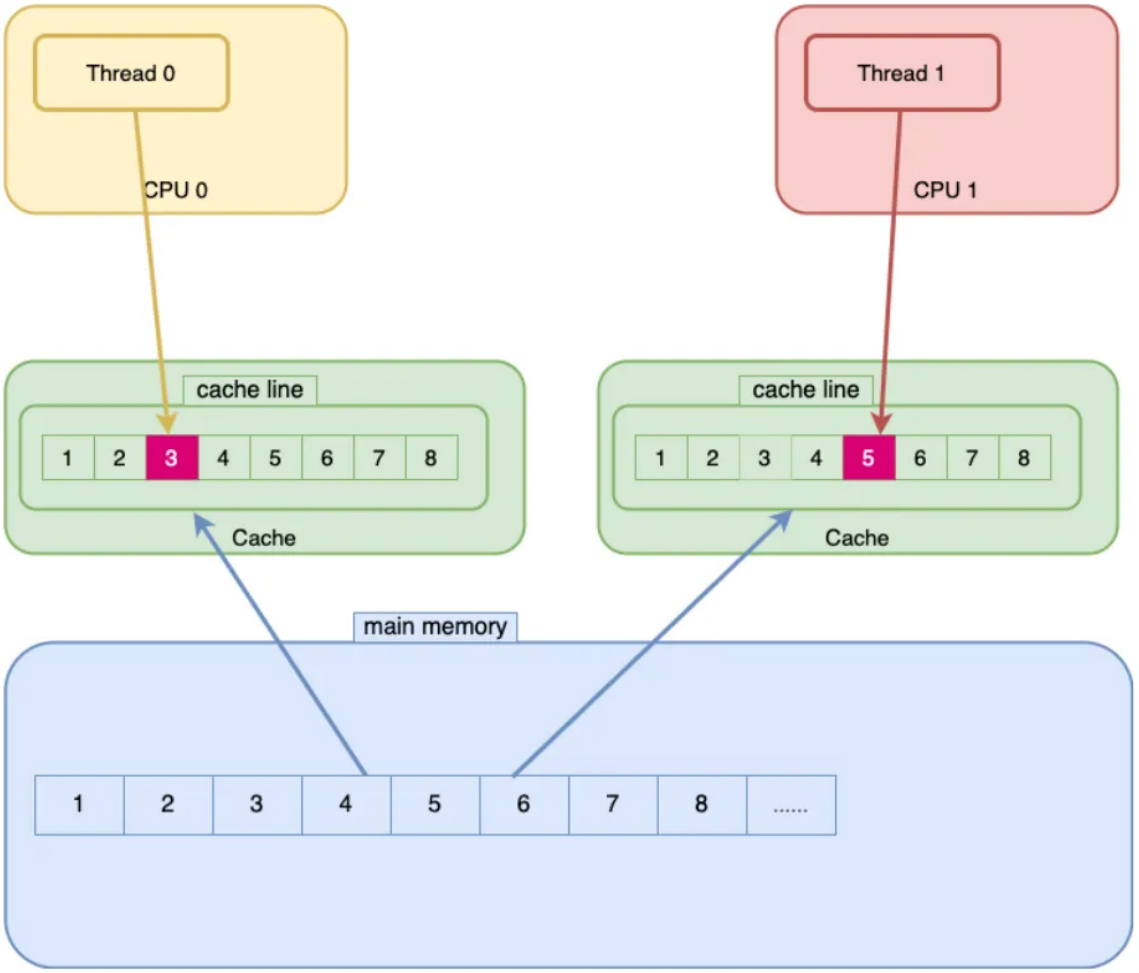
CPU缓存由cache line组成，每一个cache line为64字节，能容纳8个long值。在CPU从主存获取数据时，以cache line为单位加载，于是相邻的数据会一并加载到缓存中。很容易想到，数组的顺序遍历、相邻数据的计算是非常高效的。



2 伪共享 false sharing

CPU缓存也存在一致性问题，它通过MESI协议、MESIF协议来保证。

伪共享来源于高并发时cache line出现了缓存不一致。同一个cache line中的数据会被不同线程修改，它们相互影响，导致处理性能降低。



上图模拟一个伪共享场景，NoPadding是线程共享对象，thread0会修改no0、thread1会修改no1。当thread0修改时，除了修改自身的cache line，依据CPU缓存协议还会导致thread1对应的cache line失效，这时thread1发现cache miss后从主存加载，修改后又导致thread0的cache line失效。

```
NoPadding {
    long no0;
    long no1;
}
```

3 伪共享解决方案

padding

通过填充，让no0、no1落在不同的cache line中：

```
Padding {
    long p1, p2, p3, p4, p5, p6, p7;
    volatile long no0 = 0L;
    long p9, p10, p11, p12, p13, p14;
    volatile long no1 = 0L;
}
```

案例：jctools

Contended 注解

委托JVM填充cache line：

```
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
```

案例：JDK源码中LongAdder中的Cell、ConcurrentHashMap的CounterCell。

无锁并发

无锁并发可以从本质上解决伪共享问题，它无需填充cache line，并且执行效率是最高的，实现无锁设计的关键在于原子变量CAS（Compare-and-Swap）操作。CAS操作是一种乐观锁技术，它通过比较并交换实现无锁操作。具体来说，每个生产者或消费者线程在操作数据之前，都会先获取当前可用的元素位置，然后在该位置进行数据操作。如果在此期间，其他线程并未修改过该位置的数据（即数据未被改变），那么该线程的操作就会成功。否则，该线程就需要重试直到成功为止。

案例：disruptor

四 总结

近来于业务对接口RT提出了更高的要求，在性能优化的过程中，缓存的使用是非常多的。借此机会记录下在这段时间的思考。私以为，在引入某一项技术的时候，需整体的去看，了解其概念、原理、适用场景、注意事项，这样可以在设计之初就规避掉一些风险。

分布式缓存、本地缓存、CPU缓存涵盖的内容非常多，对细节感兴趣的同学可以阅读《Redis 设计与实现》、disruptor设计文档及代码。