zhuanlan.zhihu.com

【rocksdb源码分析】使用PinnableSlice减少Get时的内存拷贝

11-14 minutes

---

rocksdb v5.4.5版本引入一个PinnableSlice，来替换之前Get接口的出参，具体如下：

老版本：

```
Status Get(const ReadOptions& options,
                    ColumnFamilyHandle*
column_family, const Slice& key,
                    std::string* value)
```

新版本：

```
virtual Status Get(const ReadOptions& options,
                    ColumnFamilyHandle*
column_family, const Slice& key,
                    PinnableSlice* value)
```

按其说法，使用PinnableSlice* 替换std::string* 来减少

一次内存拷贝，提高读性能。我感觉挺有意思，所以看了代码，了解了下具体实现，写篇文章总结下。

1. 读流程

rocksdb读流程这里就不展开讲了，这里仅给出一次从sst文件Get的简单过程：

- DBImpl::Get()
- VersionSet::Get()
- TableCache::Get()
- BlockBasedTableReader::Get()

大的模块调用就这4步，前三步与今天主题无关，暂且忽略，最后一步BlockBasedTableReader::Get()就是从某个sst文件中读取，也是实际发生文件io，数据交换的地方，所以这一步需要详细看下，主要下面2步：

- 通过IndexBlock拿到要查找的key所在的DataBlock的Handle
- 通过这个Handle拿到对应的DataBlock，建立对应的BlockIter，seek，然后开始遍历查找

第2步关键代码如下：

```
// DataBlock的iterator
BlockIter biter;
// 通过Handle，即iiter->value()来构造biter
```

```
NewDataBlockIterator(rep_, read_options,
iiter->value(), &biter);

... ...

// seek，然后遍历查找
for (biter.Seek(key); biter.Valid();
biter.Next()) {
  ParsedInternalKey parsed_key;
  if (!ParseInternalKey(biter.key(),
&parsed_key)) {
    s = Status::Corruption(Slice());
  }

  // ---看这里，重点---
  // 数据交换的地方，如果找到，会把biter指向的数
据交换到
  // get_context的PinnableSlice
  if (!get_context->SaveValue(parsed_key,
biter.value(), &biter)) {
    done = true;
    break;
  }
}
s = biter.status();
```

```
... ...

if (done) {
  // Avoid the extra Next which is expensive in
two-level indexes
  break;
}
... ...
```

我们知道，真正数据是存在Block对象中，所以上面代码主要完成2件事，

1. 数据生成：NewDataBlockIterator()
1. 先在block_cache中查找，看有没有缓存需要的Block对象的指针，如果有就直接返回Block对象地址
2. 如果block_cache中没找到，则发生磁盘io，在sst文件中读取对应Block的内容，构造Block对象并将其地址缓存在block_cache中
2. 数据交换：get_context->SaveValue()
1. 老版本会在这里把Block中的数据拷贝到用户传进来的std::string*中
2. 新版本则是直接将Block中的数据地址赋给用户传进来的PinnableSlice*中，也就是说用户最终拿到的值的地址其实就在这个Block中。

这样就减少了一次数据拷贝，不过有一个疑问：用户拿到的值就是Block中的值，而这个Block是缓存在block_cache中的，如果后来这个Block被淘汰，那岂不是用户拿到的值被清除了？如果用引用计数来避免这个问题，那么具体怎么做呢？这个问题就是本篇的重点。

2. Cleannable

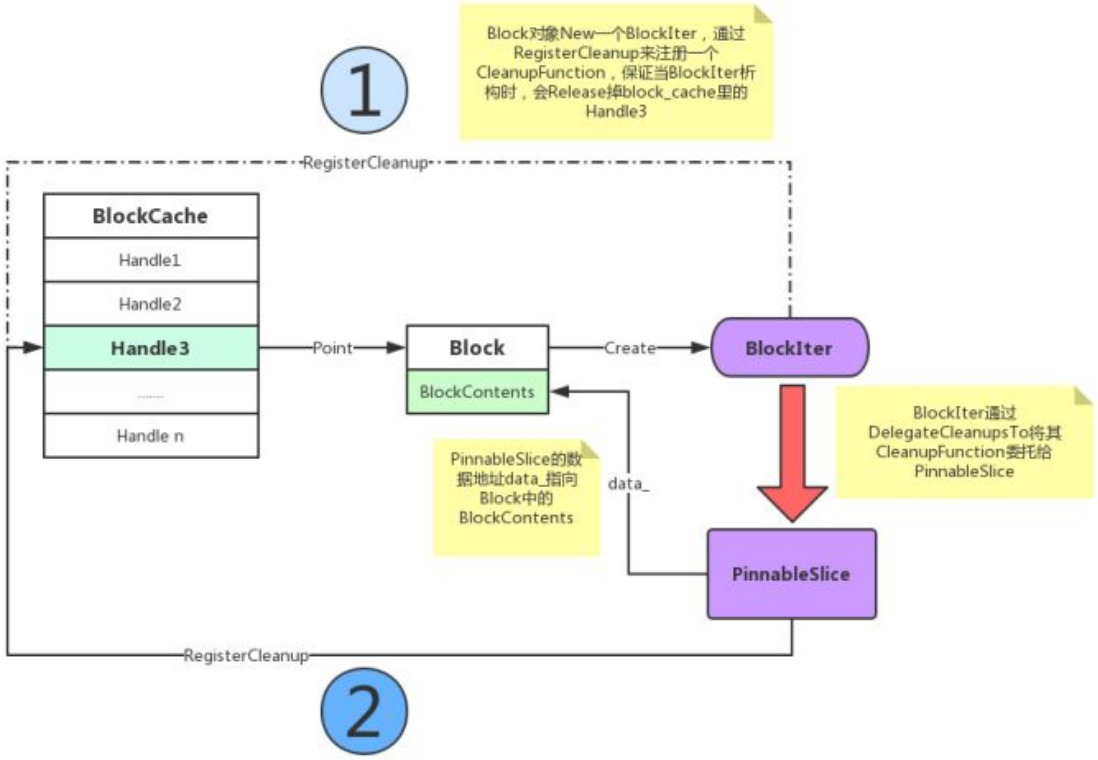
如果定义这么一个基类Cleannable，它有如下特性：

1. 可以通过RegisterCleanup方法来注册CleanupFunction，这个CleanupFunction用户自定义，一般是完成Cleannable对象申请的外部资源释放，例如释放某块之前申请的内存
2. 当其对象被析构时，会依次调用所有之前注册的CleanupFunction，来完成外部资源释放
3. 两个Cleannable子类对象A，B，可以通过A->DelegateCleanupsTo(B)，将A注册的所有CleanupFunction委托给B，这样A在析构时就不会释放它申请的外部资源，而是等到B析构时才释放

有了这么一个基类，如果将BlockIter和PinnableSlice都继承它，那么就可以BlockIter资源释放的任务委托给PinnableSlice，使得BlockIter内的资源生命周期延续至用户的PinnableSlice

原理就这个，下面详细掰扯掰扯他们之间委托了什么，

直接上图：



BlockIter肯定有其对应的Block，而Block肯定有其在block_cache中对应的Handle，所以在构造好BlockIter后，往往会执行RegisterCleanup来注册一个CleanupFunction，保证BlockIter析构时，会Release掉block_cache中的对应的Handle，而Handle被Release则会回调对应Block的deleter来最终释放Block。

BlockIter的生命周期很短，使用其在Block中查找指定key之后，他的作用域变结束。使用老版本不会有问题，因为在BlockIter作用域内，肯定会将其value拷贝给用户传入的std::string*中，而新版没有这次拷贝，用户拿到的数据地址实际就是在Block中，所以一定要想办法延长Block的作用域，不能像上面说的那样BlockIter析构便释

放。

所以新版本在BlockIter退出作用域之前，会通过DelegateCleanupsTo将其CleanupFunction委托给用户传入的PinnableSlice，这样便延长了对应block_cache中Handle的生命周期，从而延长了对应的Block的生命周期，直到用户的PinnableSlice退出作用域时，才最终回调之前BlockIter委托的CleanupFunction来进行最终的资源释放。

注：这里为了简化问题，假设block_cache的Handle在Release时一定会释放其指向的Block，实际这里会有引用计数，直到为0是才会真正释放Block

3. 总结

Cleannable基类还是挺好玩的，适合需要延长某个对象内部资源生命周期的场景。