41 | 案例分析（四）：高性能数据库连接池HiKariCP

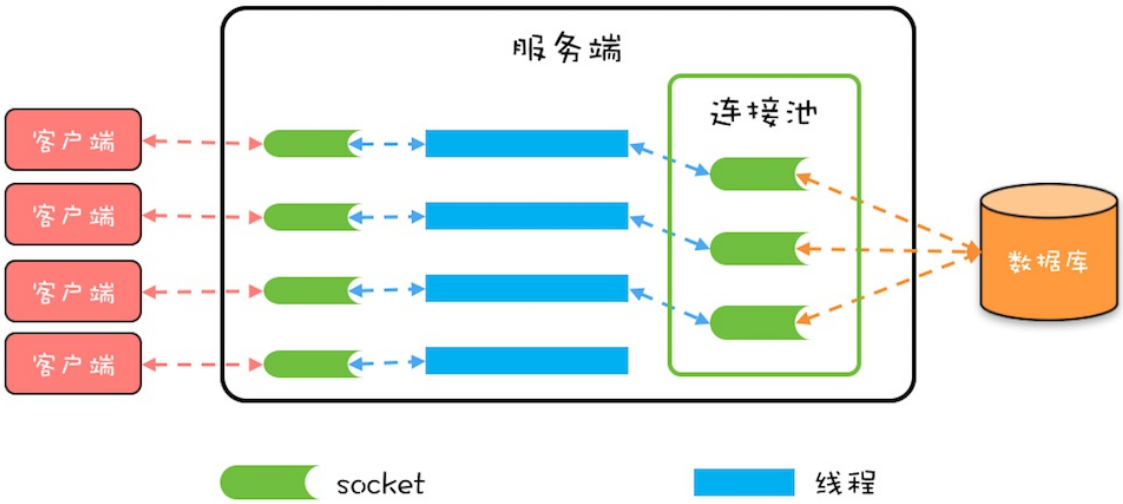
2019-06-01 王宝令

实际工作中，我们总会难免和数据库打交道；只要和数据库打交道，就免不了使用数据库连接池。业界知名的数据库连接池有不少，例如c3p0、DBCP、Tomcat JDBC Connection Pool、Druid等，不过最近最火的是HiKariCP。

HiKariCP号称是业界跑得最快的数据库连接池，这两年发展得顺风顺水，尤其是Springboot 2.0将其作为**默认数据库连接池**后，江湖一哥的地位已是毋庸置疑了。那它为什么那么快呢？今天咱们就重点聊聊这个话题。

什么是数据库连接池

在详细分析HiKariCP高性能之前，我们有必要先简单介绍一下什么是数据库连接池。本质上，数据库连接池和线程池一样，都属于池化资源，作用都是避免重量级资源的频繁创建和销毁，对于数据库连接池来说，也就是避免数据库连接频繁创建和销毁。如下图所示，服务端会在运行期持有一定数量的数据库连接，当需要执行SQL时，并不是直接创建一个数据库连接，而是从连接池中获取一个；当SQL执行完，也并不是将数据库连接真的关掉，而是将其归还到连接池中。



数据库连接池示意图

在实际工作中，我们都是使用各种持久化框架来完成数据库的增删改查，基本上不会直接和数据库连接池打交道，为了能让你更好地理解数据库连接池的工作原理，下面的示例代码并没有使用任何框架，而是原生地使用HiKariCP。执行数据库操作基本上是一系列规范化的步骤：

1. 通过数据源获取一个数据库连接；
2. 创建Statement；
3. 执行SQL；
4. 通过ResultSet获取SQL执行结果；
5. 释放ResultSet；
6. 释放Statement；
7. 释放数据库连接。

下面的示例代码，通过 ds.getConnection() 获取一个数据库连接时，其实是向数据库连接池申请一个数据库连接，而不是创建一个新的数据库连接。同样，通过 conn.close() 释放一个数据库连接时，也不是直接将连接关闭，而是将连接归还给数据库连接池。

```
//数据库连接池配置
HikariConfig config = new HikariConfig();
config.setMinimumIdle(1);
config.setMaximumPoolSize(2);
config.setConnectionTestQuery("SELECT 1");
config.setDataSourceClassName("org.h2.jdbcx.JdbcDataSource");
config.addDataSourceProperty("url", "jdbc:h2:mem:test");
// 创建数据源
DataSource ds = new HikariDataSource(config);
```

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  // 获取数据库连接
  conn = ds.getConnection();
  // 创建Statement
  stmt = conn.createStatement();
  // 执行SQL
  rs = stmt.executeQuery("select * from abc");
  // 获取结果
  while (rs.next()) {
    int id = rs.getInt(1);
    ......
  }
} catch(Exception e) {
   e.printStackTrace();
} finally {
  //关闭ResultSet
  close(rs);
  //关闭Statement
  close(stmt);
  //关闭Connection
  close(conn);
}
//关闭资源
void close(AutoCloseable rs) {
  if (rs != null) {
    try {
      rs.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

HiKariCP官方网站解释了其性能之所以如此之高的秘密。微观上HiKariCP程序编译出的字节码执

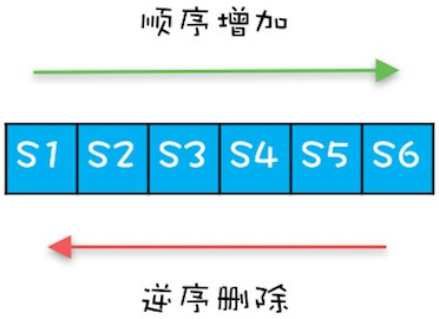
行效率更高，站在字节码的角度去优化Java代码，HiKariCP的作者对性能的执着可见一斑，不过遗憾的是他并没有详细解释都做了哪些优化。而宏观上主要是和两个数据结构有关，一个是FastList，另一个是ConcurrentBag。下面我们来看看它们是如何提升HiKariCP的性能的。

FastList解决了哪些性能问题

按照规范步骤，执行完数据库操作之后，需要依次关闭ResultSet、Statement、Connection，但是总有粗心的同学只是关闭了Connection，而忘了关闭ResultSet和Statement。为了解决这种问题，最好的办法是当关闭Connection时，能够自动关闭Statement。为了达到这个目标，Connection就需要跟踪创建的Statement，最简单的办法就是将创建的Statement保存在数组ArrayList里，这样当关闭Connection的时候，就可以依次将数组中的所有Statement关闭。

HiKariCP觉得用ArrayList还是太慢，当通过 conn.createStatement() 创建一个Statement时，需要调用ArrayList的add()方法加入到ArrayList中，这个是没有问题的；但是当通过 stmt.close() 关闭Statement的时候，需要调用 ArrayList的remove()方法来将其从ArrayList中删除，这里是有优化余地的。

假设一个Connection依次创建6个Statement，分别是S1、S2、S3、S4、S5、S6，按照正常的编码习惯，关闭Statement的顺序一般是逆序的，关闭的顺序是：S6、S5、S4、S3、S2、S1，而ArrayList的remove(Object o)方法是顺序遍历查找，逆序删除而顺序查找，这样的查找效率就太慢了。如何优化呢？很简单，优化成逆序查找就可以了。



逆序删除示意图

HiKariCP中的FastList相对于ArrayList的一个优化点就是将 remove(Object element) 方法的**查找顺序变成了逆序查找**。除此之外，FastList还有另一个优化点，是 get(int index) 方法没有对index参数进行越界检查，HiKariCP能保证不会越界，所以不用每次都进行越界检查。

整体来看，FastList的优化点还是很简单的。下面我们再来聊聊HiKariCP中的另外一个数据结构ConcurrentBag，看看它又是如何提升性能的。

ConcurrentBag解决了哪些性能问题

如果让我们自己来实现一个数据库连接池，最简单的办法就是用两个阻塞队列来实现，一个用于保存空闲数据库连接的队列idle，另一个用于保存忙碌数据库连接的队列busy；获取连接时将空闲的数据库连接从idle队列移动到busy队列，而关闭连接时将数据库连接从busy移动到idle。这种方案将并发问题委托给了阻塞队列，实现简单，但是性能并不是很理想。因为Java SDK中的阻塞队列是用锁实现的，而高并发场景下锁的争用对性能影响很大。

```
//忙碌队列
BlockingQueue<Connection> busy;
//空闲队列
BlockingQueue<Connection> idle;
```

HiKariCP并没有使用Java SDK中的阻塞队列，而是自己实现了一个叫做ConcurrentBag的并发容器。ConcurrentBag的设计最初源自C#，它的一个核心设计是使用ThreadLocal避免部分并发问题，不过HiKariCP中的ConcurrentBag并没有完全参考C#的实现，下面我们来看看它是如何实现的。

ConcurrentBag中最关键的属性有4个，分别是：用于存储所有的数据库连接的共享队列sharedList、线程本地存储threadList、等待数据库连接的线程数waiters以及分配数据库连接的工具handoffQueue。其中，handoffQueue用的是Java SDK提供的SynchronousQueue，SynchronousQueue主要用于线程之间传递数据。

```
//用于存储所有的数据库连接
CopyOnWriteArrayList<T> sharedList;
//线程本地存储中的数据库连接
ThreadLocal<List<Object>> threadList;
//等待数据库连接的线程数
AtomicInteger waiters;
//分配数据库连接的工具
SynchronousQueue<T> handoffQueue;
```

当线程池创建了一个数据库连接时，通过调用ConcurrentBag的add()方法加入到ConcurrentBag中，下面是add()方法的具体实现，逻辑很简单，就是将这个连接加入到共享队列sharedList中，如果此时有线程在等待数据库连接，那么就通过handoffQueue将这个连接分配给等待的线程。

```
//将空闲连接添加到队列
void add(final T bagEntry){
  //加入共享队列
  sharedList.add(bagEntry);
  //如果有等待连接的线程，
  //则通过handoffQueue直接分配给等待的线程
  while (waiters.get() > 0
    && bagEntry.getState() == STATE_NOT_IN_USE
    && !handoffQueue.offer(bagEntry)) {
      yield();
  }
}
```

通过ConcurrentBag提供的borrow()方法，可以获取一个空闲的数据库连接，borrow()的主要逻辑是：

1. 首先查看线程本地存储是否有空闲连接，如果有，则返回一个空闲的连接；
2. 如果线程本地存储中无空闲连接，则从共享队列中获取。
3. 如果共享队列中也没有空闲的连接，则请求线程需要等待。

需要注意的是，线程本地存储中的连接是可以被其他线程窃取的，所以需要用CAS方法防止重复分配。在共享队列中获取空闲连接，也采用了CAS方法防止重复分配。

```
T borrow(long timeout, final TimeUnit timeUnit){
  // 先查看线程本地存储是否有空闲连接
  final List<Object> list = threadList.get();
  for (int i = list.size() - 1; i >= 0; i--) {
    final Object entry = list.remove(i);
    final T bagEntry = weakThreadLocals
      ? ((WeakReference<T>) entry).get()
      : (T) entry;
    //线程本地存储中的连接也可以被窃取，
    //所以需要用CAS方法防止重复分配
    if (bagEntry != null
      && bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
      return bagEntry;
    }
```

```
    }

    // 线程本地存储中无空闲连接，则从共享队列中获取
    final int waiting = waiters.incrementAndGet();
    try {
      for (T bagEntry : sharedList) {
        //如果共享队列中有空闲连接，则返回
        if (bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
          return bagEntry;
        }
      }
      //共享队列中没有连接，则需要等待
      timeout = timeUnit.toNanos(timeout);
      do {
        final long start = currentTime();
        final T bagEntry = handoffQueue.poll(timeout, NANOSECONDS);
        if (bagEntry == null
          || bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
            return bagEntry;
        }
        //重新计算等待时间
        timeout -= elapsedNanos(start);
      } while (timeout > 10_000);
      //超时没有获取到连接，返回null
      return null;
    } finally {
      waiters.decrementAndGet();
    }
  }
```

释放连接需要调用ConcurrentBag提供的requite()方法，该方法的逻辑很简单，首先将数据库连接状态更改为STATE_NOT_IN_USE，之后查看是否存在等待线程，如果有，则分配给等待线程；如果没有，则将该数据库连接保存到线程本地存储里。

```
//释放连接
void requite(final T bagEntry){
  //更新连接状态
  bagEntry.setState(STATE_NOT_IN_USE);
  //如果有等待的线程，则直接分配给线程，无需进入任何队列
  for (int i = 0; waiters.get() > 0; i++) {
    if (bagEntry.getState() != STATE_NOT_IN_USE
      || handoffQueue.offer(bagEntry)) {
        return;
    } else if ((i & 0xff) == 0xff) {
      parkNanos(MICROSECONDS.toNanos(10));
    } else {
      yield();
    }
  }
  //如果没有等待的线程，则进入线程本地存储
  final List<Object> threadLocalList = threadList.get();
  if (threadLocalList.size() < 50) {
    threadLocalList.add(weakThreadLocals
      ? new WeakReference<>(bagEntry)
      : bagEntry);
  }
}
```

总结

HiKariCP中的FastList和ConcurrentBag这两个数据结构使用得非常巧妙，虽然实现起来并不复杂，但是对于性能的提升非常明显，根本原因在于这两个数据结构适用于数据库连接池这个特定的场景。FastList适用于逆序删除场景；而ConcurrentBag通过ThreadLocal做一次预分配，避免直接竞争共享资源，非常适合池化资源的分配。

在实际工作中，我们遇到的并发问题千差万别，这时选择合适的并发数据结构就非常重要了。当然能选对的前提是对特定场景的并发特性有深入的了解，只有了解到无谓的性能消耗在哪里，才能对症下药。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

阿健 5

同问，为什么说线程本地的连接会被窃取呢？

2019-06-01

沙漠里的骆驼 2

窃取是在获取本地链接失败时，遍历sharelist实现的

2019-06-02

峰 2

想了半天感觉ConcurrentBag应该是池化的一种通用性优化，但好像会有饥饿问题，如果某些线程总是占用连接，那么某些不经常占用连接的就可能一直拿不到连接，硬想的一个缺点，哈哈哈。

2019-06-01

晓杰 1

同问为什么线程本地的会被其他线程窃取，麻烦老师解释一下

2019-06-02

作者回复

sharedlist和其他线程的threadlocal里有可能都有同一个连接，从前者取到连接，就相当于窃取了后者

2019-06-03

空知 1

线程本地的连接会被窃取

这个我觉得是因为 如果 TI里面没有空闲的 会去 sharedList查找处于 Not_In_Use的连接 这个连接可能已经在其他TL里面存在了 所以就会出现线程T2从sharedList获取到了 T1存在TL里面存放的没有使用的连接这种情况

2019-06-02

作者回复

厉害

2019-06-03

张德 1

强烈建议老师再讲一期

2019-06-02

作者回复

呵呵

2019-06-03

cricket1981 1

可以用栈stack来代替list实现逆序关闭S6~S1吗？

2019-06-02

李海明 0

faststatementlist

2019-06-14

Geek_89bbab 0

threadList里面的连接可能也会存在于多个threadList,但是概率相对较小；threadList的连接的remove操作都由本线程来执行，窃取的线程只会把标识设置为已使用，而不会将其从对应的那个threadList移除。可能是为了避免多线程操作同一个队列,而影响性能。所以把移除threadList里的连接的任务交给对应的那个线程。

2019-06-13

QQ怪 0

根本看不够，强烈建议老师再来一篇

2019-06-03

作者回复

我觉得可以开心地笑一下，然后，就没然后了

2019-06-03

Zach_ 0

老师， 我看文中提到的是调用requite()释放链接的时候将这个链接添加到本地存储中。

那我想问，如果不是调用requite()方法释放连接的情况下，这个连接第一次被放入threadlocal是什么时候啊？ 是第一次获取连接的时候吗？

2019-06-02

作者回复

只有requite的时候会放到threatlocal里

2019-06-03

张三 0

打卡！

2019-06-02

银时空de梦 0

最后数据库连接都到线程本地池中了

2019-06-02

龙猫 0

需要多看几遍

2019-06-02

苏志辉 0

这样会不会导致每个线程持有50个以下链接，而且每个链接可能在多个线程共存

2019-06-01

粿·先生 0

面向业务设计数据结构，赞

2019-06-01

东方奇骥 0

以前只知道ArrayList删除效率低，这优化思想结合了业务场景，看起来简单不说却不知道。项目还在springboot1.x用的阿里巴巴Druid，后面新项目2.x用HaKriCP性能应该会更好。

2019-06-01

冯传博 0

线程本地的链接是如何被窃取的呢？

2019-06-01