ZLToolKit笔记8：Buffer设计

2024年12月26日　11:24

1．读缓存

```
    auto read_buffer = _poller->getSharedBuffer(sock->type() == SockNum::Sock_UDP);
```

```
SocketRecvBuffer::Ptr EventPoller::getSharedBuffer(bool is_udp) {
#if !defined(__linux) && !defined(__linux__)
    // 非Linux平台下，tcp和udp共享recvfrom方案，使用同一个buffer  [AUTO-TRANSLATED:2d2ee7bf]
    //On non-Linux platforms, tcp and udp share the recvfrom scheme, using the same buffer
    is_udp = 0;
#endif
    auto ret = _shared_buffer[is_udp].lock();
    if (!ret) {
        ret = SocketRecvBuffer::create(is_udp);
        _shared_buffer[is_udp] = ret;
    }
    return ret;
}
```

其中：

```
    //当前线程下，所有socket共享的读缓存  [AUTO-TRANSLATED:6ce70017]
    //当前线程下，所有socket共享的读缓存
// Shared read buffer for all sockets under the current thread
    std::weak_ptr<SocketRecvBuffer> _shared_buffer[2];
```

create函数：

```
static constexpr auto kPacketCount = 32;
static constexpr auto kBufferCapacity = 4 * 1024u;
SocketRecvBuffer::Ptr SocketRecvBuffer::create(bool is_udp) {
#if defined(__linux) || defined(__linux__)
    if (is_udp) {
        return std::make_shared<SocketRecvmmsgBuffer>(kPacketCount, kBufferCapacity);
    }
#endif
    return std::make_shared<SocketRecvFromBuffer>(kPacketCount * kBufferCapacity);
}
```

SocketRecvFromBuffer类的定义如下：

```
class SocketRecvFromBuffer : public SocketRecvBuffer {
public:
    SocketRecvFromBuffer(size_t size): _size(size) {}

    ssize_t recvFromSocket(int fd, ssize_t &count) override {
        ssize_t nread;
        socklen_t len = sizeof(_address);
        if (!_buffer) {
            allocBuffer();
        }
        do {
            nread = recvfrom(fd, _buffer->data(), _buffer->getCapacity() - 1, 0, (struct sockaddr
*)&_address, &len);
        } while (-1 == nread && UV_EINTR == get_uv_error(true)); //读取过程可能会被中断（EINTR），则继
```

续尝试读取

```
        if (nread > 0) {
            count = 1;  // count被赋值为1，表示读取了一次完整数据包。
            _buffer->data()[nread] = '\0';
            std::static_pointer_cast<BufferRaw>(_buffer)->setSize(nread);
        }
        return nread;
    }
    Buffer::Ptr &getBuffer(size_t index) override { return _buffer; }
    struct sockaddr_storage &getAddress(size_t index) override { return _address; }
private:
    void allocBuffer() {
        auto buf = BufferRaw::create();
        buf->setCapacity(_size);
        _buffer = std::move(buf);
    }
private:
    size_t _size;
    Buffer::Ptr _buffer;
    struct sockaddr_storage _address;
};
```

我们发现在SocketRecvFromBuffer的构造函数中并没有初始化_buffer，而只是限定了_buffer的大小_size。真正初始化_buffer的过程是在有socket可读，执行recvFromSocket函数时，通过私有的allocBuffer方法分配读缓存空间。

SocketRecvFromBuffer继承于SocketRecvBuffer：

```
class SocketRecvBuffer {
public:
    using Ptr = std::shared_ptr<SocketRecvBuffer>;
    virtual ~SocketRecvBuffer() = default;
    virtual ssize_t recvFromSocket(int fd, ssize_t &count) = 0;
    virtual Buffer::Ptr &getBuffer(size_t index) = 0;
    virtual struct sockaddr_storage &getAddress(size_t index) = 0;
    static Ptr create(bool is_udp);
};
```

但要注意：上述从socket上读取数据包时，一次只能读取一个数据包。此时count被固定为1，这也就意味着多读回调并没有真正实现(_on_mutil_read中会根据count来处理数据包）。
但如果在Linux系统下，可以采用recvmmsg系统调用，支持一次从socket读取多个数据包，此时count>1，那么多读回调就彰显出其作用了。
recvmmsg系统调用：见：

Linux系统下可由recvmmsg系统调用实现批量读取，显著提升性能。

```
#if defined(__linux) || defined(__linux__)
class SocketRecvmmsgBuffer : public SocketRecvBuffer {
public:
    SocketRecvmmsgBuffer(size_t count, size_t size)
        : _size(size)
        , _iovec(count)
        , _mmsgs(count)
        , _buffers(count)
        , _address(count) {
        for (auto i = 0u; i < count; ++i) {
            auto buf = BufferRaw::create();
            buf->setCapacity(size);
```

```
            _buffers[i] = buf;
            auto &mmsg = _mmsgs[i];
            auto &addr = _address[i];
            mmsg.msg_len = 0;
            mmsg.msg_hdr.msg_name = &addr;
            mmsg.msg_hdr.msg_namelen = sizeof(addr);
            mmsg.msg_hdr.msg_iov = &_iovec[i];
            mmsg.msg_hdr.msg_iov->iov_base = buf->data();
            mmsg.msg_hdr.msg_iov->iov_len = buf->getCapacity() - 1;
            mmsg.msg_hdr.msg_iovlen = 1;
            mmsg.msg_hdr.msg_control = nullptr;
            mmsg.msg_hdr.msg_controllen = 0;
            mmsg.msg_hdr.msg_flags = 0;
        }
    }
    ssize_t recvFromSocket(int fd, ssize_t &count) override {
        for (auto i = 0; i < _last_count; ++i) {
            auto &mmsg = _mmsgs[i];
            mmsg.msg_hdr.msg_namelen = sizeof(struct sockaddr_storage);
            auto &buf = _buffers[i];
            if (!buf) {
                auto raw = BufferRaw::create();
                raw->setCapacity(_size);
                buf = raw;
                mmsg.msg_hdr.msg_iov->iov_base = buf->data();
            }
        }
        do {
            count = recvmmsg(fd, &_mmsgs[0], _mmsgs.size(), 0, nullptr);
        } while (-1 == count && UV_EINTR == get_uv_error(true));
        _last_count = count;
        if (count <= 0) {
            return count;
        }
        ssize_t nread = 0;
        for (auto i = 0; i < count; ++i) {
            auto &mmsg = _mmsgs[i];
            nread += mmsg.msg_len;
            auto buf = std::static_pointer_cast<BufferRaw>(_buffers[i]);
            buf->setSize(mmsg.msg_len);
            buf->data()[mmsg.msg_len] = '\0';
        }
        return nread;
    }
    Buffer::Ptr &getBuffer(size_t index) override { return _buffers[index]; }
    struct sockaddr_storage &getAddress(size_t index) override { return _address[index]; }
private:
    size_t _size;
    ssize_t _last_count { 0 };
    std::vector<struct iovec> _iovec;
    std::vector<struct mmsghdr> _mmsgs;
    std::vector<Buffer::Ptr> _buffers;
    std::vector<struct sockaddr_storage> _address;
};
#endif
```

recvmmsg笔记。在套接字上接收多条消息。返回值count恰好为接收到的消息个数。

回到上述代码中recvFromSocket函数通过调用recvmmsg函数批量读取数据，读取到的内容放置在私有的成员变量_mmsgs中。这是一个元素类型为struct mmsghdr结构体的vector对象，每一个元素是一个结构体，存储着从套接字中读取到的内容。

SocketRecvmmsgBuffer 类和 SocketRecvFromBuffer 类的关系在于它们都继承自 SocketRecvBuffer，是实现 socket 数据读取逻辑的两种不同方式，主要区别在于读取方式和适用场景。

1. 两个类的核心区别

1.1 SocketRecvFromBuffer

- **核心方法**：调用 recvfrom 系统调用。
- **读取方式**：一次只从 socket 读取一个数据包。
- **适用场景**：适合普通情况下的 UDP 数据读取，逻辑简单且支持基本功能。
- **数据结构**：只需要一个缓冲区 Buffer 和一个地址结构 sockaddr_storage。
- **性能瓶颈**：每次读取需要触发一次系统调用，当数据包密集时性能可能受限。

1.2 SocketRecvmmsgBuffer

- **核心方法**：调用 recvmmsg 系统调用（Linux 特有）。
- **读取方式**：支持一次从 socket 读取多个数据包。
- **适用场景**：适合高性能需求的场景，比如处理大量短小的 UDP 数据包时，可以显著减少系统调用开销。
- **数据结构**：维护了多个缓冲区 _buffers，每个缓冲区对应一个 mmsghdr 结构。
- **性能优势**：利用 recvmmsg 实现批量读取，显著提升性能。

3. 使用场景和优势分析

3.1 SocketRecvFromBuffer 的使用场景

- 适合单次读取量较小的情况。
- 使用场景简单，调用一次只需处理一个数据包。
- 适用于普通 UDP 应用，如间隔较大的数据包处理。

3.2 SocketRecvmmsgBuffer 的使用场景

- 适合高吞吐量场景。
- 适用于短时间内大量数据包涌入的情况，例如：
  - 视频流传输。
  - 网络游戏中频繁的状态同步。
  - 高并发日志收集。

3.3 优势对比

特性	SocketRecvFromBuffer	SocketRecvmmsgBuffer
系统调用	recvfrom	recvmmsg（批量调用）
数据包读取	单个数据包	多个数据包
性能	一般	高（减少系统调用开销）
适用场景	普通场景	高吞吐量场景
平台支持	跨平台（POSIX 标准）	Linux 专有

4. 总结

- SocketRecvFromBuffer 和 SocketRecvmmsgBuffer 都是 SocketRecvBuffer 的子类，分别针对单包读取和多

包批量读取的需求。

- SocketRecvFromBuffer：
  - 实现简单，适合常规场景。
  - 每次调用 recvfrom，系统调用频率高。
- SocketRecvmmsgBuffer：
  - 利用 recvmmsg 实现批量读取，性能更优。
  - 适用于高性能场景，尤其是处理高密度、小数据包的 UDP 应用。
- 如果在 Linux 平台上运行且性能是关键，可以优先使用 SocketRecvmmsgBuffer。否则，SocketRecvFromBuffer 是更通用的选择。

在SocketRecvFromBuffer和SocketRecvmmsgBuffer中都封装了Buffer对象。
它们两都实现了一个成员方法recvFromSocket用以从socket中读取数据到buffer中；
如果此时buffer尚未分配将会首先进行分配内存：

```
        for (auto i = 0; i < _last_count; ++i) {
            auto &mmsg = _mmsgs[i];
            mmsg.msg_hdr.msg_namelen = sizeof(struct sockaddr_storage);
            auto &buf = _buffers[i];
            if (!buf) {
                auto raw = BufferRaw::create();
                raw->setCapacity(_size);
                buf = raw;
                mmsg.msg_hdr.msg_iov->iov_base = buf->data();
            }
        }
```

调用的是BufferRaw（它是Buffer的派生类）中的静态成员方法create()，定义如下：

```
BufferRaw::Ptr BufferRaw::create() {
#if 0
    static ResourcePool<BufferRaw> packet_pool;
    static onceToken token([]() {
        packet_pool.setSize(1024);
    });
    auto ret = packet_pool.obtain2();
    ret->setSize(0);
    return ret;
#else
    return Ptr(new BufferRaw);
#endif
}
```

它将构造一个BufferRaw的对象，用智能指针管理，并返回这个智能指针。

```
BufferRaw(size_t capacity = 0) {
        if (capacity) {
            setCapacity(capacity);
        }
    }
```

然后调用setCapacity(_size)方法来设定读缓存大小。

2. 写缓存

在ZLToolKit笔记7：TCP通信中。简单阐述了有关TCP通信中发送数据时的两级写缓存和互斥锁：

```
    // 一级发送缓存，socket可写时，会把一级缓存批量送入到二级缓存 [AUTO-TRANSLATED:26f1da58]
```

```
    //First-level send cache, when the socket is writable, it will batch the first-level cache into the second-level cache
    List<std::pair<Buffer::Ptr, bool>> _send_buf_waiting;
    // 一级发送缓存锁 [AUTO-TRANSLATED:9ec6c6a9]
    //First-level send cache lock
    MutexWrapper<std::recursive_mutex> _mtx_send_buf_waiting;
    // 二级发送缓存，socket可写时，会把二级缓存批量写入到socket [AUTO-TRANSLATED:cc665665]
    //Second-level send cache, when the socket is writable, it will batch the second-level cache into the socket
    List<BufferList::Ptr> _send_buf_sending;
    // 二级发送缓存锁 [AUTO-TRANSLATED:306e3472]
    //Second-level send cache lock
    MutexWrapper<std::recursive_mutex> _mtx_send_buf_sending;
```

在笔记ZLToolKit笔记7：TCP通信中已经在flushData函数中，阐述了当二级发送缓存为空，且一级发送缓存非空时会将一级发送缓存中的内容移动到二级发送缓存中（List的移动构造）。
在BufferList中的静态成员函数BufferList::create 方法用于创建缓存对象。它将返回一个管理BufferList对象的智能指针。

```
class BufferList : public noncopyable {
public:
    using Ptr = std::shared_ptr<BufferList>;
    using SendResult = std::function<void(const Buffer::Ptr &buffer, bool send_success)>;
    BufferList() = default;
    virtual ~BufferList() = default;
    virtual bool empty() = 0;
    virtual size_t count() = 0;
    virtual ssize_t send(int fd, int flags) = 0;
    static Ptr create(List<std::pair<Buffer::Ptr, bool> > list, SendResult cb, bool is_udp);
private:
    //对象个数统计 [AUTO-TRANSLATED:3b43e8c2]
    //Object count statistics
    ObjectStatistic<BufferList> _statistic;
};
```

create函数中将根据系统条件和是否是udp包生成具体的派生类对象，并返回：

```
BufferList::Ptr BufferList::create(List<std::pair<Buffer::Ptr, bool> > list, SendResult cb, bool is_udp)
{
#if defined(_WIN32)
    if (is_udp) {
        // sendto/send 方案，待优化 [AUTO-TRANSLATED:e94184aa]
        //sendto/send scheme, to be optimized
        return std::make_shared<BufferSendTo>(std::move(list), std::move(cb), is_udp);
    }
    // WSASend方案 [AUTO-TRANSLATED:9ac7bb81]
    //WSASend scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#elif defined(__linux__) || defined(__linux)
    if (is_udp) {
        // sendmmsg方案 [AUTO-TRANSLATED:4596c2c4]
        //sendmmsg scheme
        return std::make_shared<BufferSendMMsg>(std::move(list), std::move(cb));
    }
```

```
    // sendmsg方案 [AUTO-TRANSLATED:8846f9c4]
    //sendmsg scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#else
    if (is_udp) {
        // sendto/send 方案，可优化? [AUTO-TRANSLATED:21dbae7c]
        //sendto/send scheme, can be optimized?
        return std::make_shared<BufferSendTo>(std::move(list), std::move(cb), is_udp);
    }
    // sendmsg方案 [AUTO-TRANSLATED:8846f9c4]
    //sendmsg scheme
    return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
#endif
}
```

以上面标黄的为例：将构建一个BufferSendMsg的对象并用智能指针管理，BufferSendMsg声明如下：

```
class BufferSendMsg final : public BufferList, public BufferCallBack {
public:
    using SocketBufVec = std::vector<SocketBuf>;
    BufferSendMsg(List<std::pair<Buffer::Ptr, bool> > list, SendResult cb);
    ~BufferSendMsg() override = default;
    bool empty() override;
    size_t count() override;
    ssize_t send(int fd, int flags) override;
private:
    void reOffset(size_t n);
    ssize_t send_l(int fd, int flags);
private:
    size_t _iovec_off = 0;
    size_t _remain_size = 0;
    SocketBufVec _iovec;
};
```

该类继承于BufferList和BufferCallBack。

真正准备要发送的二级缓存中的数据在BufferCallBack类中所定义：

```
class BufferCallBack {
public:
    BufferCallBack(List<std::pair<Buffer::Ptr, bool> > list, BufferList::SendResult cb)
        : _cb(std::move(cb))
        , _pkt_list(std::move(list)) {}
    ~BufferCallBack() {
        sendCompleted(false);
    }
    void sendCompleted(bool flag) {
        if (_cb) {
            //全部发送成功或失败回调 [AUTO-TRANSLATED:6b9a9abf]
            //All send success or failure callback
            while (!_pkt_list.empty()) {
                _cb(_pkt_list.front().first, flag);
                _pkt_list.pop_front();
            }
        } else {
            _pkt_list.clear();
        }
    }
    void sendFrontSuccess() {
        if (_cb) {
```

```
            //发送成功回调  [AUTO-TRANSLATED:52759efc]
            //Send success callback
            _cb(_pkt_list.front().first, true);
        }
        _pkt_list.pop_front();
    }
protected:
    BufferList::SendResult _cb;
    List<std::pair<Buffer::Ptr, bool> > _pkt_list;
};
```

在一级缓存向二级缓存批量移动时，调用的其实是List的移动构造函数，这样会使得一级缓存中内容不复存在。

这个过程封装在了基类BufferList:: create的调用过程中了，如：

```
return std::make_shared<BufferSendMsg>(std::move(list), std::move(cb));
```

的调用了派生类BufferSendMsg的构造函数：

```
BufferSendMsg::BufferSendMsg(List<std::pair<Buffer::Ptr, bool>> list, SendResult cb)
    : BufferCallBack(std::move(list), std::move(cb))
    , _iovec(_pkt_list.size()) {
    auto it = _iovec.begin();
    _pkt_list.for_each([&](std::pair<Buffer::Ptr, bool> &pr) {
#if !defined(_WIN32)
        it->iov_base = pr.first->data();
        it->iov_len = pr.first->size();
        _remain_size += it->iov_len;
#else
        it->buf = pr.first->data();
        it->len = pr.first->size();
        _remain_size += it->len;
#endif
        ++it;
    });
}
```

在BufferSendMsg构造过程中首先会构造基类对象。

BufferCallBack的定义上面有。在它的构造函数中：

```
BufferCallBack(List<std::pair<Buffer::Ptr, bool> > list, BufferList::SendResult cb)
        : _cb(std::move(cb))
        , _pkt_list(std::move(list)) {}
```

其中

```
    List<std::pair<Buffer::Ptr, bool> > _pkt_list;
```

而且传入的参数list即为一级缓存类型也为：

```
    List<std::pair<Buffer::Ptr, bool>> _send_buf_waiting;
```

在BufferSendMsg构造过程中（基类对象构造完毕之后）会对_iovec这个成员变量做初始化，也就是个根据二级缓存中的内容刚刚移动到_pkt_list中，初始化即将二级缓存中内容拷贝到_iovec这个变量中，以便在后续发送。

```
    SocketBufVec _iovec;
```

其中：

```
using SocketBufVec = std::vector<SocketBuf>;

#if defined(_WIN32)
using SocketBuf = WSABUF;
#else
using SocketBuf = iovec;
#endif
```

也就是在Linux系统下，**_iovec这个变量是个元素类型为struct iovec的vector对象。**

不论在发送数据还是接收数据时，调用recvmsg/sendmsg：

```
#include<sys/socket.h>
ssize_t recvmsg(int sockfd,struct msghdr*msg,int flags);
ssize_t sendmsg(int sockfd,struct msghdr*msg,int flags);
```

其中msghdr这个结构体中就封装了struct iovec：

```
struct msghdr
{
    void*msg_name;/*socket地址*/
    socklen_t msg_namelen;/*socket地址的长度*/
    struct iovec*msg_iov;/*分散的内存块，见后文*/
    int msg_iovlen;/*分散内存块的数量*/
    void*msg_control;/*指向辅助数据的起始位置*/
    socklen_t msg_controllen;/*辅助数据的大小*/
    int msg_flags;/*复制函数中的flags参数，并在调用过程中更新*/
};
```

这样如何从二级发送缓存是如何设计与保存数据的整个逻辑也就清晰了。

3. Buffer设计

不论是读还是写都会用到缓存，ZLToolKit的设计中同一个poller线程监听的socket共享读缓存；而写缓存是有两级，每个socket单独具有这两级的发送缓存。

本节不管写的过程是如何实现的，单纯考虑写buffer是如何实现的。

（1）首先来看Buffer：

```
namespace toolkit {

template <typename T> struct is_pointer : public std::false_type {};
template <typename T> struct is_pointer<std::shared_ptr<T> > : public std::true_type {};
template <typename T> struct is_pointer<std::shared_ptr<T const> > : public std::true_type {};
template <typename T> struct is_pointer<T*> : public std::true_type {};
template <typename T> struct is_pointer<const T*> : public std::true_type {};

//缓存基类  [AUTO-TRANSLATED:d130ab72]
//Cache base class
class Buffer : public noncopyable {…

template <typename C>
class BufferOffset : public  Buffer {…

using BufferString = BufferOffset<std::string>;

//指针式缓存对象，  [AUTO-TRANSLATED:c8403290]
//Pointer-style cache object,
class BufferRaw : public Buffer {…

class BufferLikeString : public Buffer {…

}//namespace toolkit
#endif //ZLTOOLKIT_BUFFER_H
```

```
//缓存基类  [AUTO-TRANSLATED:d130ab72]
//Cache base class
class Buffer : public noncopyable {
public:
```

```
    using Ptr = std::shared_ptr<Buffer>;
    Buffer() = default;
    virtual ~Buffer() = default;
    //返回数据长度 [AUTO-TRANSLATED:955f731c]
    //Return data length
    virtual char *data() const = 0;
    virtual size_t size() const = 0;
    virtual std::string toString() const {
        return std::string(data(), size());
    }
    virtual size_t getCapacity() const {
        return size();
    }
private:
    //对象个数统计 [AUTO-TRANSLATED:3b43e8c2]
    //Object count statistics
    ObjectStatistic<Buffer> _statistic;
};
```

不论在上面介绍的读缓存还是两级的写缓存，最后都是基于Buffer实现的。
从第一部分1中可以看到读缓存的建立是调用BufferRaw::create()实现的。

```
//指针式缓存对象， [AUTO-TRANSLATED:c8403290]
//Pointer-style cache object,
class BufferRaw : public Buffer {
public:
    using Ptr = std::shared_ptr<BufferRaw>;
    static Ptr create();
    ~BufferRaw() override {
        if (_data) {
            delete[] _data;
        }
    }
    //在写入数据时请确保内存是否越界 [AUTO-TRANSLATED:5602043e]
    //When writing data, please ensure that the memory does not overflow
    char *data() const override {
        return _data;
    }
    //有效数据大小 [AUTO-TRANSLATED:b8dcbda7]
    //Effective data size
    size_t size() const override {
        return _size;
    }
    //分配内存大小 [AUTO-TRANSLATED:cce87adf]
    //Allocated memory size
    void setCapacity(size_t capacity) {
        。。。
    }
    //设置有效数据大小 [AUTO-TRANSLATED:efc4fb3e]
    //Set valid data size
    virtual void setSize(size_t size) {
        if (size > _capacity) {
            throw std::invalid_argument("Buffer::setSize out of range");
        }
        _size = size;
    }
```

```
    //赋值数据 [AUTO-TRANSLATED:0b91b213]
    //Assign data
    void assign(const char *data, size_t size = 0) {
        if (size <= 0) {
            size = strlen(data);
        }
        setCapacity(size + 1);
        memcpy(_data, data, size);
        _data[size] = '\0';
        setSize(size);
    }
    size_t getCapacity() const override {
        return _capacity;
    }
protected:
    friend class ResourcePool_l<BufferRaw>;
    BufferRaw(size_t capacity = 0) {
        if (capacity) {
            setCapacity(capacity);
        }
    }
    BufferRaw(const char *data, size_t size = 0) {
        assign(data, size);
    }
private:
    size_t _size = 0;
    size_t _capacity = 0;
    char *_data = nullptr;
    //对象个数统计 [AUTO-TRANSLATED:3b43e8c2]
    //Object count statistics
    ObjectStatistic<BufferRaw> _statistic;
};

BufferRaw::Ptr BufferRaw::create() {
#if 0
    static ResourcePool<BufferRaw> packet_pool;
    static onceToken token([]() {
        packet_pool.setSize(1024);
    });
    auto ret = packet_pool.obtain2();
    ret->setSize(0);
    return ret;
#else
    return Ptr(new BufferRaw);
#endif
}
```

BufferRaw的设计体现了工厂模式的精髓：

（1）提供了统一的对象创建接口create；

（2）内部封装了对象创建逻辑，包括直接分配和对象池（内存池）复用；

（3）调用者只需依赖接口，不关注实现细节；

（4）利用工厂模式提高了代码的可扩展性、复用性和性能。

这里有关工厂模式的学习参见ZLToolKit笔记9：设计模式

在BufferRaw中最为关键的成员函数便是setCapacity了：

```
//分配内存大小 [AUTO-TRANSLATED:cce87adf]
```

```
//Allocated memory size
void setCapacity(size_t capacity) {
    if (_data) {
        do {
            if (capacity > _capacity) {
                //请求的内存大于当前内存，那么重新分配 [AUTO-TRANSLATED:65306424]
                //If the requested memory is greater than the current memory, reallocate
                break;
            }
            if (_capacity < 2 * 1024) {
                //2K以下，不重复开辟内存，直接复用 [AUTO-TRANSLATED:056416c0]
                //Less than 2K, do not repeatedly allocate memory, reuse directly
                return;
            }
            if (2 * capacity > _capacity) {
                //如果请求的内存大于当前内存的一半，那么也复用 [AUTO-TRANSLATED:c189d660]
                //If the requested memory is greater than half of the current memory, also reuse
                return;
            }
        } while (false);
        delete[] _data;
    }
    _data = new char[capacity];
    _capacity = capacity;
}
```

读写缓存的内存空间的频繁分配与释放会影响性能开销，可以在BufferRaw::create方法中可以选择BufferRaw对象的创建逻辑通过预先分配好的对象池（资源池/内存池）。有关该项目中内存池的详解参见：ZLToolKit笔记10：内存池。