课前须知

课程讲授的内容

包括：

图灵学院-Redis进阶课程表	
章节	章节名称
1	Redis队列Stream、Redis多线程详解
2	Redis HyperLogLog与事务、Redis 7.0前瞻

上课说明：

课程前置知识：第四期 Redis、第五期 Redis 前面章节、Linux 基本概念、基本操作、基本的 SSM、SpringBoot 等。

1、首次出现的知识如需要进行编码，一般会进行手写，以后再出现则可能会事先准备好或者进行拷贝。

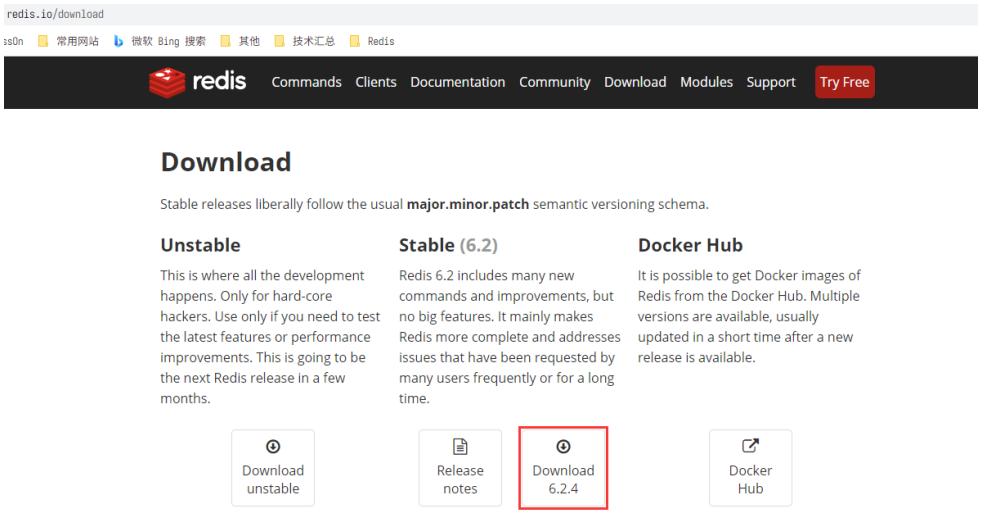
2、一个知识点如果大部分同学明白，不会重复讲解，未明白的同学请看视频、笔记、请教同学或加老师 QQ。

3、以上为本课的章节安排，**不是课时安排**，如果一章内容在一次课内未讲完，则会顺延到后面的课程继续讲解。

课程 Redis 版本选择和安装

Redis 目前最新版本为 Redis-6.2.6 ，考虑到实际的情况，本次课程会以 CentOS7 下 Redis-6.2.4 版本进行讲解。

下载地址：https://redis.io/download



Installation

From source code

Download, extract and compile Redis with:

```
$ wget https://download.redis.io/releases/redis-6.2.4.tar.gz
$ tar xzf redis-6.2.4.tar.gz
$ cd redis-6.2.4
$ make
```

安装运行 Redis 很简单，在 Linux 下执行上面的 4 条命令即可 ，同时前面的课程已经有完整的视频讲解，请到网盘中下载观看，并自行安装。如安装过程出错，请保证安装包完整无误,依赖包无误，并仔细阅读安装错误日志和检查操作系统层面的用户、用户组、文件和目录是否存在，各种权限是否正确等！

同时 Redis 的官方文档相当丰富和齐全，WEB 地址如下：

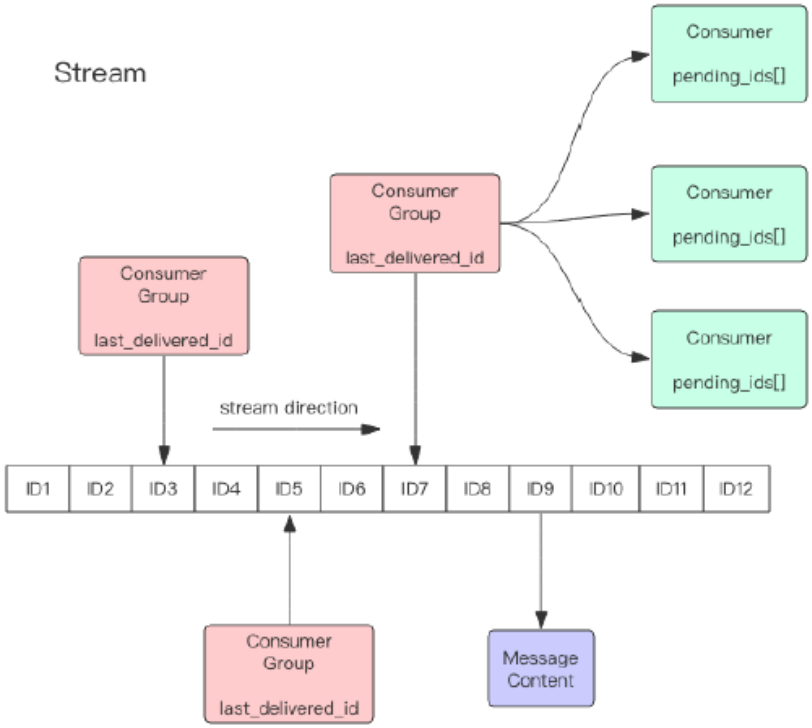
https://redis.io/documentation

Redis 队列与 Stream、Redis 6 多线程详解

Redis 队列与 Stream

Redis5.0 最大的新特性就是多出了一个数据结构 Stream，它是一个新的强大的支持多播的可持久化的消息队列，作者声明 Redis Stream 地借鉴了 Kafka 的设计。

Stream 总述



Redis Stream 的结构如上图所示,每一个Stream都有一个消息链表，将所有加入的消息都串起来,每个消息都有一个唯一的 ID 和对应的内容。消息是持久化的，Redis 重启后，内容还在。

每个 Stream 都有唯一的名称，它就是 Redis 的 key，在我们首次使用xadd指令追加消息时自动创建。

每个 Stream 都可以挂多个消费组，每个消费组会有个游标last_delivered_id在 Stream 数组之上往前移动，表示当前消费组已经消费到哪条消息了。每个消费组都有一个 Stream 内唯一的名称，消费组不会自动创建，它需要单独的指令xgroup create进行创建，需要指定从 Stream 的某个消息 ID 开始消费，这个 ID 用来初始化last_delivered_id变量。

每个消费组 (Consumer Group) 的状态都是独立的，相互不受影响。也就是说同一份 Stream 内部的消息会被每个消费组都消费到。

同一个消费组 (Consumer Group) 可以挂接多个消费者 (Consumer)，这些消费者之间是竞争关系，任意一个消费者读取了消息都会使游标last_delivered_id往前移动。每个消费者有一个组内唯一名称。

消费者 (Consumer) 内部会有个状态变量 pending_ids，它记录了当前已经被客户端读取,但是还没有 ack 的消息。如果客户端没有 ack，这个变量里面的消息 ID 会越来越多，一旦某个消息被 ack，它就开始减少。这个 pending_ids 变量在 Redis 官方被称之为 PEL，也就是 Pending Entries List，这是一个很核心的数据结构，它用来确保客户端至少消费了消息一次，而不会在网络传输的中途丢失了没处理。

消息 ID 的形式是 timestampInMillis-sequence，例如 1527846880572-5，它表示当前的消息在毫米时间戳 1527846880572 时产生,并且是该毫秒内产生的第

5 条消息。消息 ID 可以由服务器自动生成，也可以由客户端自己指定，但是形式必须是整数-整数，而且必须是后面加入的消息的 ID 要大于前面的消息 ID。

消息内容就是键值对，形如 hash 结构的键值对，这没什么特别之处。

常用操作命令

生产端

xadd 追加消息

xdel 删除消息，这里的删除仅仅是设置了标志位，不会实际删除消息。

xrange 获取消息列表，会自动过滤已经删除的消息

xlen 消息长度

del 删除 Stream

xadd streamtest * name mark age 18

```
127.0.0.1:6880> xadd streamtest * name mark age 18
"1626705954593-0"
```

streamtest 表示当前这个队列的名字，也就是我们一般意义上 Redis 中的 key，* 号表示服务器自动生成 ID，后面顺序跟着“name mark age 18”，是我们存入当前 streamtest 这个队列的消息，采用的也是 key/value 的存储形式

返回值 1626705954593-0 则是生成的消息 ID，由两部分组成：时间戳-序号。时间戳时毫秒级单位，是生成消息的 Redis 服务器时间，它是个 64 位整型。序号是在这个毫秒时间点内的消息序号。它也是个 64 位整型。

为了保证消息是有序的，因此 Redis 生成的 ID 是单调递增有序的。由于 ID 中包含时间戳部分，为了避免服务器时间错误而带来的问题（例如服务器时间延后了），Redis 的每个 Stream 类型数据都维护一个 latest_generated_id 属性，用于记录最后一个消息的 ID。若发现当前时间戳退后（小于 latest_generated_id 所记录的），则采用时间戳不变而序号递增的方案来作为新消息 ID（这也是序号为什么使用 int64 的原因，保证有足够多的的序号），从而保证 ID 的单调递增性质。

如果不是非常特别的需求，强烈建议使用 Redis 的方案生成消息 ID，因为这种时间戳+序号的单调递增的 ID 方案，几乎可以满足全部的需求，但 ID 是支持自定义的。

```
127.0.0.1:6880> xadd streamtest * name james age 23
"1626706380924-0"
127.0.0.1:6880> xadd streamtest * name king age 19
"1626706393957-0"
127.0.0.1:6880> xlen streamtest
(integer) 3
```

xrange streamtest - +

其中-表示最小值 ，+ 表示最大值

```
127.0.0.1:6880> xrange streamtest - +
1) 1) "1626705954593-0"
   2) 1) "name"
      2) "mark"
      3) "age"
      4) "18"
2) 1) "1626706380924-0"
   2) 1) "name"
      2) "james"
      3) "age"
      4) "23"
3) 1) "1626706393957-0"
   2) 1) "name"
      2) "king"
      3) "age"
      4) "19"
```

或者我们可以指定消息 ID 的列表：

```
127.0.0.1:6880> xrange streamtest - 1626706380924-0
1) 1) "1626705954593-0"
   2) 1) "name"
      2) "mark"
      3) "age"
      4) "18"
2) 1) "1626706380924-0"
   2) 1) "name"
      2) "james"
      3) "age"
      4) "23"
127.0.0.1:6880> xrange streamtest 1626706380924-0 +
1) 1) "1626706380924-0"
   2) 1) "name"
      2) "james"
      3) "age"
      4) "23"
2) 1) "1626706393957-0"
   2) 1) "name"
      2) "king"
      3) "age"
      4) "19"
```

xdel streamtest 1626706380924-0

xlen streamtest

```
127.0.0.1:6880> xdel streamtest 1626706380924-0
(integer) 1
127.0.0.1:6880> xlen streamtest
(integer) 2
127.0.0.1:6880>  xrange streamtest - +
1) 1) "1626705954593-0"
   2) 1) "name"
      2) "mark"
      3) "age"
      4) "18"
2) 1) "1626706393957-0"
   2) 1) "name"
      2) "king"
      3) "age"
      4) "19"
```

del streamtest　删除整个 Stream

```
127.0.0.1:6880> del streamtest
(integer) 1
```

消费端

单消费者

虽然 Stream 中有消费者组的概念，但是可以在不定义消费组的情况下进行 Stream 消息的独立消费，当 Stream 没有新消息时，甚至可以阻塞等待。Redis 设计了一个单独的消费指令 xread，可以将 Stream 当成普通的消息队列 (list) 来使用。使用 xread 时，我们可以完全忽略消费组 (Consumer Group) 的存在，就好比 Stream 就是一个普通的列表 (list)。

xread count 1 streams stream2 0-0

“count 1”表示从 Stream 读取 1 条消息，缺省当然是头部，“streams”可以理解为 Redis 关键字，“stream2”指明了要读取的队列名称，“0-0”指从头开始

```
127.0.0.1:6880> xread count 1 streams stream2 0-0
1) 1) "stream2"
   2) 1) 1) "1626710882927-0"
         2) 1) "name"
            2) "king"
            3) "age"
            4) "19"
```

xread count 2 streams stream2 1626710882927-0

也可以指定从 streams 的消息 Id 开始(不包括命令中的消息 id)

```
127.0.0.1:6880> xread count 2 streams stream2 1626710882927-0
1) 1) "stream2"
   2) 1) 1) "1626710894233-0"
         2) 1) "name"
            2) "mark"
            3) "age"
            4) "18"
      2) 1) "1626710901961-0"
         2) 1) "name"
            2) "james"
            3) "age"
            4) "32"
```

xread count 1 streams stream2 $

$代表从尾部读取，上面的意思就是从尾部读取最新的一条消息,此时默认不返回任何消息

```
127.0.0.1:6880> xread count 1 streams stream2 $
(nil)
```

所以最好以阻塞的方式读取尾部最新的一条消息，直到新的消息的到来

xread block 0 count 1 streams stream2 $

block 后面的数字代表阻塞时间，单位毫秒

```
127.0.0.1:6880> xread block 0 count 1 streams stream2 $
```

此时我们新开一个客户端，往 stream2 中写入一条消息

```
[redis@iZwz9j203ithc4guluvb2wZ src]$ ./redis-cli -p
127.0.0.1:6880> xadd stream2 * dafei 23
"1626748062536-0"
```

可以看到阻塞解除了，返回了新的消息内容，而且还显示了一个等待时间，这里我们等待了 127.87s

```
127.0.0.1:6880> xread block 0 count 1 streams stream2 $
1) 1) "stream2"
   2) 1) 1) "1626748062536-0"
         2) 1) "dafei"
            2) "23"
(127.97s)
```

一般来说客户端如果想要使用 xread 进行顺序消费，一定要记住当前消费到哪里了，也就是返回的消息 ID。下次继续调用 xread 时，将上次返回的最后一个消息 ID 作为参数传递进去，就可以继续消费后续的消息。

消费组

创建消费组

Stream 通过 xgroup create 指令创建消费组 (Consumer Group)，需要传递起始消息 ID 参数用来初始化 last_delivered_id 变量。

xgroup create stream2 cg1 0-0

“stream2”指明了要读取的队列名称，“cg1”表示消费组的名称，“0-0”表示从头开始消费

```
127.0.0.1:6880> xgroup create stream2 cg1 0-0
OK
```

xgroup create stream2 cg2 $

$ 表示从尾部开始消费，只接受新消息，当前 Stream 消息会全部忽略

```
127.0.0.1:6880> xgroup create stream2 cg2 $
OK
```

现在我们可以用 xinfo 命令来看看 stream2 的情况：

xinfo stream stream2

```
127.0.0.1:6880> xinfo stream stream2
 1) "length"
 2) (integer) 4
 3) "radix-tree-keys"
 4) (integer) 1
 5) "radix-tree-nodes"
 6) (integer) 2
 7) "last-generated-id"
 8) "1626748062536-0"
 9) "groups"
10) (integer) 2
11) "first-entry"
12) 1) "1626710882927-0"
    2) 1) "name"
       2) "king"
       3) "age"
       4) "19"
13) "last-entry"
14) 1) "1626748062536-0"
    2) 1) "dafei"
       2) "23"
```

消息长度

最后生成的消息id

2个消费组

xinfo groups stream2

```
127.0.0.1:6880> xinfo groups stream2
1) 1) "name"
   2) "cg1"
   3) "consumers"
   4) (integer) 0
   5) "pending"
   6) (integer) 0
   7) "last-delivered-id"
   8) "0-0"
2) 1) "name"
   2) "cg2"
   3) "consumers"
   4) (integer) 0
   5) "pending"
   6) (integer) 0
   7) "last-delivered-id"
   8) "1626748062536-0"
```

消费组的情况

消息消费

有了消费组，自然还需要消费者，Stream 提供了 xreadgroup 指令可以进行消费组的组内消费，需要提供消费组名称、消费者名称和起始消息 ID。

它同 xread 一样，也可以阻塞等待新消息。读到新消息后，对应的消息 ID 就会进入消费者的 PEL(正在处理的消息) 结构里，客户端处理完毕后使用 xack 指令通知服务器，本条消息已经处理完毕，该消息 ID 就会从 PEL 中移除。

xreadgroup GROUP cg1 c1 count 1 streams stream2 >

“GROUP”属于关键字，“cg1”是消费组名称，“c1”是消费者名称，“count 1”指明了消费数量，> 号表示从当前消费组的 last_delivered_id 后面开始读，每当消费者读取一条消息，last_delivered_id 变量就会前进

```
127.0.0.1:6880> xreadgroup GROUP cg1 c1 count 1 streams stream2 >
1) 1) "stream2"
   2) 1) 1) "1626710882927-0"
         2) 1) "name"
            2) "king"
            3) "age"
            4) "19"
```

前面我们定义 cg1 的时候是从头开始消费的，自然就获得 Stream2 中第一条消息

再执行一次上面的命令

```
            4) "19"
127.0.0.1:6880> xreadgroup GROUP cg1 c1 count 1 streams stream2 >
1) 1) "stream2"
   2) 1) 1) "1626710894233-0"
         2) 1) "name"
            2) "mark"
            3) "age"
            4) "18"
```

自然就读取到了下条消息。

我们将 Stream2 中的消息读取完

xreadgroup GROUP cg1 c1 count 2 streams stream2 >

很自然就没有消息可读了， **xreadgroup GROUP cg1 c1 count 1 streams stream2 >**

```
127.0.0.1:6880> xreadgroup GROUP cg1 c1 count 2 streams stream2 >
1) 1) "stream2"
   2) 1) 1) "1626710901961-0"
         2) 1) "name"
            2) "james"
            3) "age"
            4) "32"
      2) 1) "1626748062536-0"
         2) 1) "dafei"
            2) "23"
127.0.0.1:6880> xreadgroup GROUP cg1 c1 count 1 streams stream2 >
(nil)
```

然后设置阻塞等待

xreadgroup GROUP cg1 c1 block 0 count 1 streams stream2 >

```
127.0.0.1:6880> xreadgroup GROUP cg1 c1 block 0 count 1 streams stream2 >
```

我们新开一个客户端，发送消息到 stream2

xadd stream2 * name lison score 98

```
127.0.0.1:6880> xadd stream2 * name lison score 98
"1626751586744-0"
```

回到原来的客户端，发现阻塞解除，收到新消息

```
127.0.0.1:6880> xreadgroup GROUP cg1 c1 block 0 count 1 streams stream2 >
1) 1) "stream2"
   2) 1) 1) "1626751586744-0"
         2) 1) "name"
            2) "lison"
            3) "score"
            4) "98"
(296.75s)
```

我们来观察一下观察消费组状态

```
127.0.0.1:6880> xinfo groups stream2
1) 1) "name"
   2) "cg1"
   3) "consumers"
   4) (integer) 1
   5) "pending"
   6) (integer) 5
   7) "last-delivered-id"
   8) "1626751586744-0"
2) 1) "name"
   2) "cg2"
   3) "consumers"
   4) (integer) 0
   5) "pending"
   6) (integer) 0
   7) "last-delivered-id"
   8) "1626748062536-0"
```

一个消费者

有5条消息未ACK

群组2没有任何变化

如果同一个消费组有多个消费者，我们还可以通过 xinfo consumers 指令观察每个消费者的状态

xinfo consumers stream2 cg1

```
127.0.0.1:6880> xinfo consumers stream2 cg1
1) 1) "name"
   2) "c1"
   3) "pending"
   4) (integer) 5
   5) "idle"
   6) (integer) 441340
```

可以看到目前 c1 这个消费者有 5 条待 ACK 的消息，空闲了 441340 ms 没有读取消息。

如果我们确认一条消息

xack stream2 cg1 1626751586744-0

就可以看到待确认消息变成了 4 条

```
127.0.0.1:6880> xinfo consumers stream2 cg1
1) 1) "name"
   2) "c1"
   3) "pending"
   4) (integer) 4
   5) "idle"
   6) (integer) 815915
```

xack 允许带多个消息 id，比如

```
127.0.0.1:6880> xack stream2 cg1 1626710901961-0 1626748062536-0
(integer) 2
```

同时 Stream 还提供了命令 XPENDIING 用来获消费组或消费内消费者的未处理完毕的消息，每个 Pending 的消息有 4 个属性：

消息 ID

所属消费者

IDLE，已读取时长

delivery counter，消息被读取次数

命令 XCLAIM 用以进行消息转移的操作，将某个消息转移到自己的 Pending 列表中。需要设置组、转移的目标消费者和消息 ID，同时需要提供 IDLE（已被读取时长），只有超过这个时长，才能被转移。

更多的 Redis 的 Stream 命令请大家参考 Redis 官方文档：

https://redis.io/topics/streams-intro

https://redis.io/commands

同时 Redis 文档中，在每个命令的详情页右边会显示“Related commands”，可以通过这个列表快速了解相关的命令和进入具体命令的详情页。

Redis 队列几种实现的总结

基于 List 的 LPUSH+BRPOP 的实现

足够简单，消费消息延迟几乎为零，但是需要处理空闲连接的问题。

如果线程一直阻塞在那里，Redis 客户端的连接就成了闲置连接，闲置过久，服务器一般会主动断开连接，减少闲置资源占用，这个时候 blpop 和 brpop 或抛出异常，所以在编写客户端消费者的时候要小心，如果捕获到异常需要重试。

其他缺点包括：

做消费者确认 ACK 麻烦，不能保证消费者消费消息后是否成功处理的问题（宕机或处理异常等），通常需要维护一个 Pending 列表，保证消息处理确认；不能做广播模式，如 pub/sub，消息发布/订阅模型；不能重复消费，一旦消费就会被删除；不支持分组消费。

基于 Sorted-Set 的实现

多用来实现延迟队列，当然也可以实现有序的普通的消息队列，但是消费者无法阻塞的获取消息，只能轮询，不允许重复消息。

PUB/SUB，订阅/发布模式

优点：

典型的广播模式，一个消息可以发布到多个消费者；多信道订阅，消费者可以同时订阅多个信道，从而接收多类消息；消息即时发送，消息不用等待消费者读取，消费者会自动接收到信道发布的消息。

缺点：

消息一旦发布，不能接收。换句话就是发布时若客户端不在线，则消息丢失，不能寻回；不能保证每个消费者接收的时间是一致的；若消费者客户端出现消息积压，到一定程度，会被强制断开，导致消息意外丢失。通常发生在消息的生产远大于消费速度时；可见，Pub/Sub 模式不适合做消息存储，消息积压类的业务，而是擅长处理广播，即时通讯，即时反馈的业务。

基于 Stream 类型的实现

基本上已经有了一个消息中间件的雏形，可以考虑在生产过程中使用，当然真正要在生产中应用，要做的事情还很多，比如消息队列的管理和监控就需要花大力气去实现，而专业消息队列都已经自带或者存在着很好的第三方方案和插件。

与 Java 的集成

可以参见 cn.tuling.redis.redismq.StreamVer

消息队列问题

从我们上面对 Stream 的使用表明，Stream 已经具备了一个消息队列的基本要素，生产者 API、消费者 API，消息 Broker，消息的确认机制等等，所以在使用消息中间件中产生的问题，这里一样也会遇到。

Stream 消息太多怎么办?

要是消息积累太多，Stream 的链表岂不是很长，内容会不会爆掉?xdel 指令又不会删除消息，它只是给消息做了个标志位。

Redis 自然考虑到了这一点，所以它提供了一个定长 Stream 功能。在 xadd 的指令提供一个定长长度 maxlen，就可以将老的消息干掉，确保最多不超过指定长度。

消息如果忘记 ACK 会怎样?

Stream 在每个消费者结构中保存了正在处理中的消息 ID 列表 PEL，如果消费者收到了消息处理完了但是没有回复 ack，就会导致 PEL 列表不断增长，如果有很多消费组的话，那么这个 PEL 占用的内存就会放大。所以消息要尽可能的快速消费并确认。

PEL 如何避免消息丢失?

在客户端消费者读取 Stream 消息时，Redis 服务器将消息回复给客户端的过程中，客户端突然断开了连接，消息就丢失了。但是 PEL 里已经保存了发出去的消息 ID。待客户端重新连上之后，可以再次收到 PEL 中的消息 ID 列表。

不过此时 xreadgroup 的起始消息 ID 不能为参数>，而必须是任意有效的消息 ID，一般将参数设为 0-0，表示读取所有的 PEL 消息以及自 last_delivered_id 之后的新消息。

死信问题

如果某个消息，不能被消费者处理，也就是不能被 XACK，这是要长时间处于 Pending 列表中，即使被反复的转移给各个消费者也是如此。此时该消息的 delivery counter（通过 XPENDING 可以查询到）就会累加，当累加到某个我们预设的临界值时，我们就认为是坏消息（也叫死信，DeadLetter，无法投递的消息），由于有了判定条件，我们将坏消息处理掉即可，删除即可。删除一个消息，使用 XDEL 语法，注意，这个命令并没有删除 Pending 中的消息，因此查看 Pending，消息还会在，可以在执行执行 XDEL 之后，XACK 这个消息标识其处理完毕。

Stream 的高可用

Stream 的高可用是建立主从复制基础上的，它和其它数据结构的复制机制没有区别，也就是说在 Sentinel 和 Cluster 集群环境下 Stream 是可以支持高可用的。不过鉴于 Redis 的指令复制是异步的，在 failover 发生时，Redis 可能会丢失极小部分数据，这点 Redis 的其它数据结构也是一样的。

分区 Partition

Redis 的服务器没有原生支持分区能力，如果想要使用分区，那就需要分配多个 Stream，然后在客户端使用一定的策略来生产消息到不同的 Stream。

Stream 小结

Stream 的消费模型借鉴了 Kafka 的消费分组的概念，它弥补了 Redis Pub/Sub 不能持久化消息的缺陷。但是它又不同于 kafka，Kafka 的消息可以分 partition，而 Stream 不行。如果非要分 parition 的话，得在客户端做，提供不同的 Stream 名称，对消息进行 hash 取模来选择往哪个 Stream 里塞。

总的来说，如果是中小项目和企业，在工作中已经使用了 Redis，在业务量不是很大，而又需要消息中间件功能的情况下，可以考虑使用 Redis 的 Stream 功能。但是如果并发量很高，资源足够支持下，还是以专业的消息中间件，比如 RocketMQ、Kafka 等来支持业务更好。

Redis 中的线程和 IO 模型

什么是 Reactor 模式 ？

“反应”器名字中”反应“的由来：

“反应”即“倒置”，“控制逆转”,具体事件处理程序不调用反应器，而向反应器注册一个事件处理器，表示自己对某些事件感兴趣，有时间来了，具体

事件处理程序通过事件处理器对某个指定的事件发生做出反应；这种控制逆转又称为“好莱坞法则”（不要调用我，让我来调用你）

例如，路人甲去做男士 SPA，前台的接待小姐接待了路人甲，路人甲现在只对 10000 技师感兴趣，但是路人甲去的比较早，就告诉接待小姐，等 10000 技师上班了或者是空闲了，通知我。等路人甲接到接待小姐通知，做出了反应，把 10000 技师占住了。

然后，路人甲想起上一次的那个 10000 号房间不错，设备舒适，灯光暧昧，又告诉前台的接待小姐，我对 10000 号房间很感兴趣，房间空出来了就告诉我，我现在先和 10000 这个小姐聊下人生，10000 号房间空出来了，路人甲再次接到接待小姐通知，路人甲再次做出了反应。

路人甲就是具体事件处理程序，前台的接待小姐就是所谓的反应器，“10000 技师上班了”和“10000 号房间空闲了”就是事件，路人甲只对这两个事件感兴趣，其他，比如 10001 号技师或者 10002 号房间空闲了也是事件，但是路人甲不感兴趣。

前台的接待小姐不仅仅服务路人甲 1 人，他还可以同时服务路人乙、丙……，每个人所感兴趣的事件是不一样的，前台的接待小姐会根据每个人感兴趣的事件通知对应的每个人。

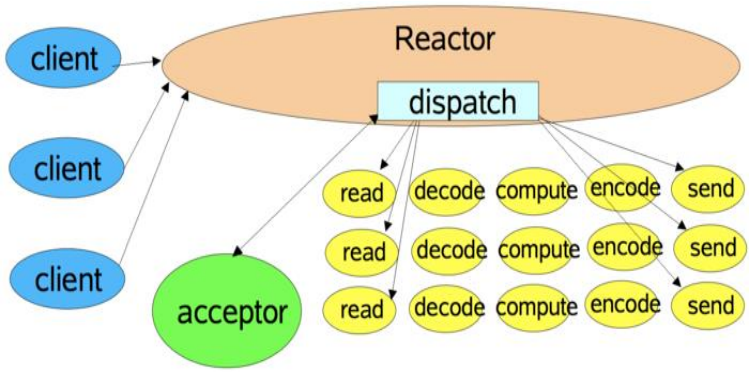
单线程 Reactor 模式流程

服务器端的 Reactor 是一个线程对象，该线程会启动事件循环，并使用 Acceptor 事件处理器关注 ACCEPT 事件，这样 Reactor 会监听客户端向服务器端发起的连接请求事件(ACCEPT 事件)。

客户端向服务器端发起一个连接请求，Reactor 监听到了该 ACCEPT 事件的发生并将该 ACCEPT 事件派发给相应的 Acceptor 处理器来进行处理。建立连接后关注的 READ 事件，这样一来 Reactor 就会监听该连接的 READ 事件了。

当 Reactor 监听到有读 READ 事件发生时，将相关的事件派发给对应的处理器进行处理。比如，读处理器会通过读取数据，此时 read()操作可以直接读取到数据，而不会堵塞与等待可读的数据到来。

在目前的单线程 Reactor 模式中，不仅 I/O 操作在该 Reactor 线程上，连非 I/O 的业务操作也在该线程上进行处理了，这可能会大大延迟 I/O 请求的响应。所以我们应该将非 I/O 的业务逻辑操作从 Reactor 线程上卸载，以此来加速 Reactor 线程对 I/O 请求的响应。



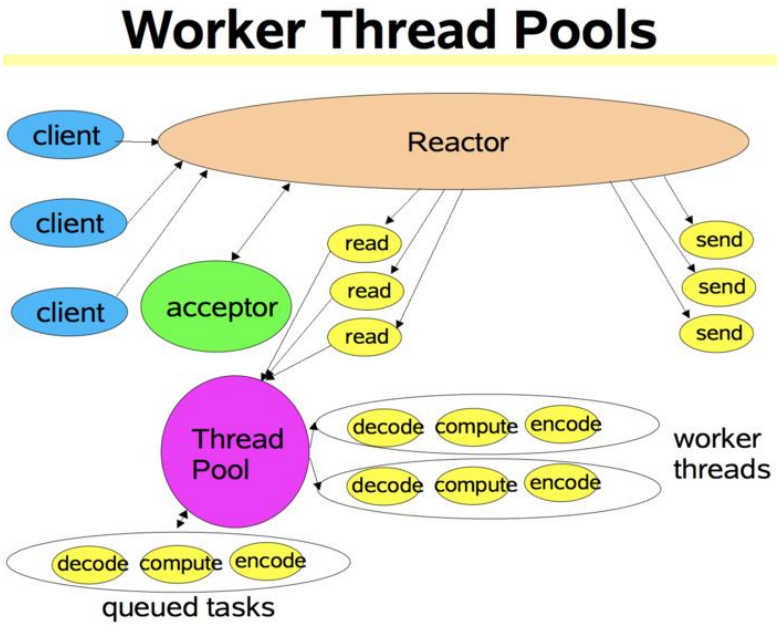
单线程 Reactor，工作者线程池

与单线程 Reactor 模式不同的是，添加了一个工作者线程池，并将非 I/O 操作从 Reactor 线程中移出转交给工作者线程池来执行。这样能够提高 Reactor 线程的 I/O 响应，不至于因为一些耗时的业务逻辑而延迟对后面 I/O 请求的处理。

但是对于一些小容量应用场景，可以使用单线程模型，对于高负载、大并发或大数据量的应用场景却不合适，主要原因如下：

① 一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的读取和发送；

② 当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈；

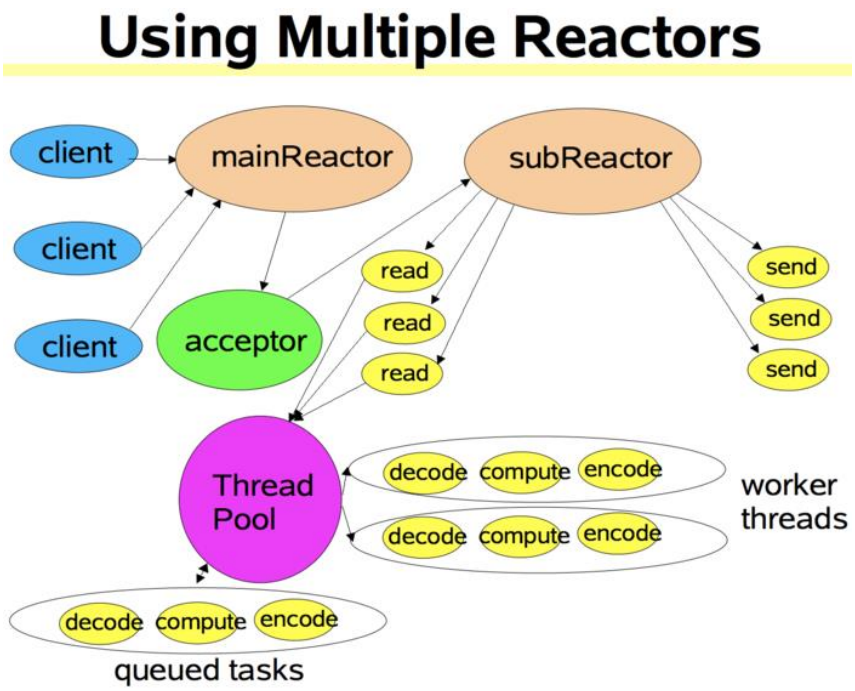


多 Reactor 线程模式

Reactor 线程池中的每一 Reactor 线程都会有自己的 Selector、线程和分发的事件循环逻辑。

mainReactor 可以只有一个，但 subReactor 一般会有多个。mainReactor 线程主要负责接收客户端的连接请求，然后将接收到的 SocketChannel 传递给 subReactor，由 subReactor 来完成和客户端的通信。

多 Reactor 线程模式将“接受客户端的连接请求”和“与该客户端的通信”分在了两个 Reactor 线程来完成。mainReactor 完成接收客户端连接请求的操作，它不负责与客户端的通信，而是将建立好的连接转交给 subReactor 线程来完成与客户端的通信，这样一来就不会因为 read()数据量太大而导致后面的客户端连接请求得不到即时处理的情况。并且多 Reactor 线程模式在海量的客户端并发请求的情况下，还可以通过实现 subReactor 线程池来将海量的连接分发给多个 subReactor 线程，在多核的操作系统中这能大大提升应用的负载和吞吐量。



Redis 中的线程和 IO 概述

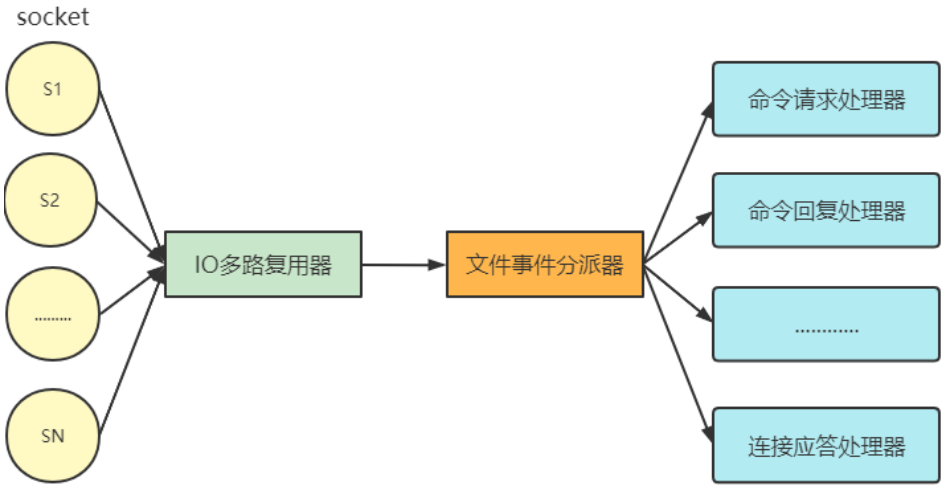
Redis 基于 Reactor 模式开发了自己的网络事件处理器 - 文件事件处理器（file event handler，后文简称为 FEH），而该处理器又是单线程的，所以 redis 设计为单线程模型。

采用 I/O 多路复用同时监听多个 socket，根据 socket 当前执行的事件来为 socket 选择对应的事件处理器。

当被监听的 socket 准备好执行 accept、read、write、close 等操作时，和操作对应的文件事件就会产生，这时 FEH 就会调用 socket 之前关联好的事件处理器来处理对应事件。

所以虽然 FEH 是单线程运行，但通过 I/O 多路复用监听多个 socket，不仅实现高性能的网络通信模型，又能和 Redis 服务器中其它同样单线程运行的模块交互，保证了 Redis 内部单线程模型的简洁设计。

下面来看文件事件处理器的几个组成部分。



socket

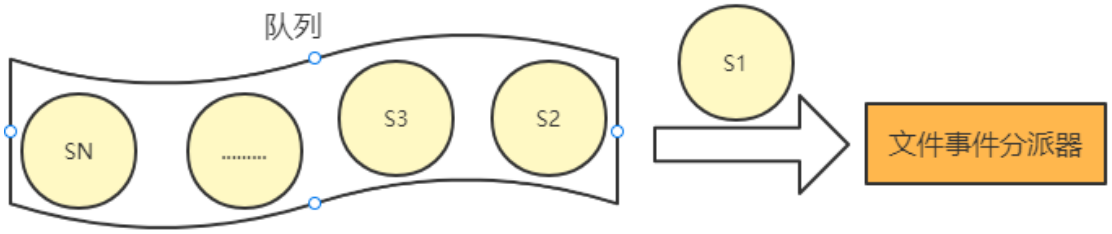
文件事件就是对 socket 操作的抽象， 每当一个 socket 准备好执行连接 accept、read、write、close 等操作时， 就会产生一个文件事件。一个服务器通常会连接多个 socket， 多个 socket 可能并发产生不同操作，每个操作对应不同文件事件。

I/O 多路复用程序

I/O 多路复用程序会负责监听多个 socket。

尽管文件事件可能并发出现， 但 I/O 多路复用程序会将所有产生事件的 socket 放入队列， 通过该队列以有序、同步且每次一个 socket 的方式向文件事件分派器传送 socket。

当上一个 socket 产生的事件被对应事件处理器执行完后， I/O 多路复用程序才会向文件事件分派器传送下个 socket， 如下：



I/O 多路复用程序的实现

Redis 的 I/O 多路复用程序的所有功能都是通过包装常见的 select、epoll、evport 和 kqueue 这些 I/O 多路复用函数库实现的。

每个 I/O 多路复用函数库在 Redis 源码中都对应一个单独的文件：

因为 Redis 为每个 I/O 多路复用函数库都实现了相同的 API ， 所以 I/O 多路复用程序的底层实现是可以互换的。Redis 在 I/O 多路复用程序的实现源码 `ae.c`文件中宏定义了相应规则，使得程序在编译时自动选择系统中性能最高的 I/O 多路复用函数库作为 Redis 的 I/O 多路复用程序的底层实现：性能降序排列。

```
#ifdef HAVE_EVPORT
#include "ae_evport.c"
#else
    #ifdef HAVE_EPOLL
    #include "ae_epoll.c"
    #else
        #ifdef HAVE_KQUEUE
        #include "ae_kqueue.c"
        #else
        #include "ae_select.c"
        #endif
    #endif
#endif
```

注：

evport = Solaris 10

epoll = Linux

kqueue = OS X，FreeBSD

select =通常作为fallback 安装在所有平台上

Evport，Epoll 和 KQueue 具有 O(1)描述符选择算法复杂度，并且它们都使用内部内核空间内存结构.他们还可以提供很多(数十万个)文件描述符.

除其他外，select 最多只能提供 1024 个描述符，并且对描述符进行完全扫描(因此每次迭代所有描述符以选择一个可使用的描述符)，因此复杂性是 O(n).

文件事件分派器

文件事件分派器接收 I/O 多路复用程序传来的 socket， 并根据 socket 产生的事件类型， 调用相应的事件处理器。

文件事件处理器

服务器会为执行不同任务的套接字关联不同的事件处理器， 这些处理器是一个个函数， 它们定义了某个事件发生时， 服务器应该执行的动作。

Redis 为各种文件事件需求编写了多个处理器，若客户端连接 Redis，对连接服务器的各个客户端进行应答，就需要将 socket 映射到连接应答处理器写数据到 Redis，接收客户端传来的命令请求，就需要映射到命令请求处理器从 Redis 读数据，向客户端返回命令的执行结果，就需要映射到命令回复处理器当主服务器和从服务器进行复制操作时， 主从服务器都需要映射到特别为复制功能编写的复制处理器。

文件事件的类型

I/O 多路复用程序可以监听多个 socket 的 ae.h/AE_READABLE 事件和 ae.h/AE_WRITABLE 事件， 这两类事件和套接字操作之间的对应关系如下：

当 socket 可读（比如客户端对 Redis 执行 write/close 操作），或有新的可应答的 socket 出现时（即客户端对 Redis 执行 connect 操作），socket 就会产生一个 AE_READABLE 事件。

当 socket 可写时（比如客户端对 Redis 执行 read 操作），socket 会产生一个 AE_WRITABLE 事件。

I/O 多路复用程序可以同时监听 AE_REABLE 和 AE_WRITABLE 两种事件，要是一个 socket 同时产生这两种事件，那么文件事件分派器优先处理 AE_REABLE 事件。即一个 socket 又可读又可写时， Redis 服务器先读后写 socket。

总结

最后，让我们梳理一下客户端和 Redis 服务器通信的整个过程：

Redis 启动初始化时，将连接应答处理器跟 AE_READABLE 事件关联。

若一个客户端发起连接，会产生一个 AE_READABLE 事件，然后由连接应答处理器负责和客户端建立连接，创建客户端对应的 socket，同时将这个 socket 的 AE_READABLE 事件和命令请求处理器关联，使得客户端可以向主服务器发送命令请求。

当客户端向 Redis 发请求时（不管读还是写请求），客户端 socket 都会产生一个 AE_READABLE 事件，触发命令请求处理器。处理器读取客户端的命令内容，然后传给相关程序执行。

当 Redis 服务器准备好给客户端的响应数据后，会将 socket 的 AE_WRITABLE 事件和命令回复处理器关联，当客户端准备好读取响应数据时，会在 socket 产生一个 AE_WRITABLE 事件，由对应命令回复处理器处理，即将准备好的响应数据写入 socket，供客户端读取。

命令回复处理器全部写完到 socket 后，就会删除该 socket 的 AE_WRITABLE 事件和命令回复处理器的映射。

Redis6 中的多线程

1. Redis6.0 之前的版本真的是单线程吗?

Redis 在处理客户端的请求时，包括获取 (socket 读)、解析、执行、内容返回 (socket 写) 等都由一个顺序串行的主线程处理，这就是所谓的“单线程”。但如果严格来讲从 Redis4.0 之后并不是单线程，除了主线程外，它也有后台线程在处理一些较为缓慢的操作，例如清理脏数据、无用连接的释放、大 key 的删除等等。

2. Redis6.0 之前为什么一直不使用多线程?

官方曾做过类似问题的回复：使用 Redis 时，几乎不存在 CPU 成为瓶颈的情况， Redis 主要受限于内存和网络。例如在一个普通的 Linux 系统上，Redis 通过

使用 pipelining 每秒可以处理 100 万个请求，所以如果应用程序主要使用 O(N)或 O(log(N))的命令，它几乎不会占用太多 CPU。

使用了单线程后，可维护性高。多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，带来了并发读写的一系列问题，增加了系统复杂度、同时可能存在线程切换、甚至加锁解锁、死锁造成的性能损耗。Redis 通过 AE 事件模型以及 IO 多路复用等技术，处理性能非常高，因此没有必要使用多线程。单线程机制使得 Redis 内部实现的复杂度大大降低，Hash 的惰性 Rehash、Lpush 等等 “线程不安全” 的命令都可以无锁进行。

3. Redis6.0 为什么要引入多线程呢?

Redis 将所有数据放在内存中，内存的响应时长大约为 100 纳秒，对于小数据包，Redis 服务器可以处理 80,000 到 100,000 QPS，这也是 Redis 处理的极限了，对于 80%的公司来说，单线程的 Redis 已经足够使用了。

但随着越来越复杂的业务场景，有些公司动不动就上亿的交易量，因此需要更大的 QPS。常见的解决方案是在分布式架构中对数据进行分区并采用多个服务器，但该方案有非常大的缺点，例如要管理的 Redis 服务器太多，维护代价大；某些适用于单个 Redis 服务器的命令不适用于数据分区；数据分区无法解决热点读/写问题；数据偏斜，重新分配和放大/缩小变得更加复杂等等。

从 Redis 自身角度来说，因为读写网络的 read/write 系统调用占用了 Redis 执行期间大部分 CPU 时间，瓶颈主要在于网络的 IO 消耗，优化主要有两个方向:

- 提高网络 IO 性能，典型的实现比如使用 DPDK 来替代内核网络栈的方式
- 使用多线程充分利用多核，典型的实现比如 Memcached。

协议栈优化的这种方式跟 Redis 关系不大，支持多线程是一种最有效最便捷的操作方式。所以总结起来，redis 支持多线程主要就是两个原因：

- 可以充分利用服务器 CPU 资源，目前主线程只能利用一个核
- 多线程任务可以分摊 Redis 同步 IO 读写负荷

4.Redis6.0 默认是否开启了多线程?

Redis6.0 的多线程默认是禁用的，只使用主线程。如需开启需要修改 redis.conf 配置文件：io-threads-do-reads yes

```
# io-threads 4
#
# Setting io-threads to 1 w
# When I/O threads are enab
# to thread the write(2) sy
# socket. However it is als
# protocol parsing using th
# it to yes:
#
# io-threads-do-reads no
#
```

开启多线程后，还需要设置线程数，否则是不生效的。

关于线程数的设置，官方有一个建议：4 核的机器建议设置为 2 或 3 个线程，8 核的建议设置为 6 个线程，线程数一定要小于机器核数。还需要注意的是，线程数并不是越大越好，官方认为超过了 8 个基本就没什么意义了。

5.Redis6.0 采用多线程后，性能的提升效果如何?

Redis 作者 antirez 在 RedisConf 2019 分享时曾提到：Redis 6 引入的多线程 IO 特性对性能提升至少是一倍以上。国内也有大牛曾使用 unstable 版本在阿里云 esc 进行过测试，GET/SET 命令在 4 线程 IO 时性能相比单线程是几乎是翻倍了。如果开启多线程，至少要 4 核的机器，且 Redis 实例已经占用相当大的 CPU 耗时的时候才建议采用，否则使用多线程没有意义。

6.Redis6.0 多线程的实现机制?

流程简述如下：

1、主线程负责接收建立连接请求，获取 socket 放入全局等待读处理队列

2、主线程处理完读事件之后，通过 RR(Round Robin) 将这些连接分配给这些 IO 线程

3、主线程阻塞等待 IO 线程读取 socket 完毕

4、主线程通过单线程的方式执行请求命令，请求数据读取并解析完成，但并不执行回写 socket

5、主线程阻塞等待 IO 线程将数据回写 socket 完毕

6、解除绑定，清空等待队列

该设计有如下特点：

1、IO 线程要么同时在读 socket，要么同时在写，不会同时读或写

2、IO 线程只负责读写 socket 解析命令，不负责命令处理

7.开启多线程后，是否会存在线程并发安全问题?

从上面的实现机制可以看出，Redis 的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程顺序执行。所以我们不需要去考虑控制 key、lua、事务，LPUSH/LPOP 等等的并发及线程安全问题。

8.Redis6.0 的多线程和 Memcached 多线程模型进行对比

Memcached 服务器采用 master-woker 模式进行工作，服务端采用 socket 与客户端通讯。主线程、工作线程 采用 pipe 管道进行通讯。主线程采用 libevent 监听 listen、accept 的读事件，事件响应后将连接信息的数据结构封装起来，根据算法选择合适的工作线程，将连接任务携带连接信息分发出去，相应的线程利用连接描述符建立与客户端的 socket 连接 并进行后续的存取数据操作。

相同点：都采用了 master 线程-worker 线程的模型

不同点：Memcached 执行主逻辑也是在 worker 线程里，模型更加简单，实现了真正的线程隔离，符合我们对线程隔离的常规理解。而 Redis 把处理逻辑交还给 master 线程，虽然一定程度上增加了模型复杂度，但也解决了线程并发安全等问题

本文档分享地址

http://note.youdao.com/noteshare?id=76f9bf31a33fac0422af26242b26c14a&sub=28454BAED81A4BF5B3EBA2EAB35C0DA4