redis
Clients

Java SpringBoot New
- Introduction
- Strings
- Lists
- Sets
- ZSets
- Hashes
- Streams
- Common Keys
- Pipelining
- Pub/Sub
- Master/Replica
- Sentinel
- Cluster
- ---
- Auto Config
- Manual Config
- Load Balancing (readFrom)
- RedisTemplate
- Connection Pool & Thread
- Async Spring & Lettuce
- DB select
- Spring Multi Data Source
- Lettuce Multi Data Source
- Spring Project Create
- Spring Project Eclipse
- Spring Project IntelliJ
- ---
- Spring Session Standalone
- Spring Session MasterRepli
- Spring Session Sentinel
- Spring Session Cluster

Java Lettuce(Spring)New

Java Lettuce(Plain)

Java Jedis

Java Redisson

C Hiredis

C# StackExchange

PHP PhpRedis

PHP Predis

Redis Admin & Monitoring Tool

Spring Data Redis Pipelining

Spring Data Redis Pipelining

1. Redis Pipelining (Redis 문서)

Redis 명령을 일괄 처리하여 왕복 시간을 최적화하는 방법
How to optimize round-trip times by batching Redis commands

Redis 파이프라이닝은 개별 명령에 대한 응답을 기다리지 않고 한 번에 여러 명령을 실행하여 성능을 향상시키는 기술입니다. 파이프라이닝은 대부분의 Redis 클라이언트에서 지원됩니다. 이 문서에서는 파이프라이닝이 해결하도록 설계된 문제와 Redis에서 파이프라이닝이 작동하는 방식을 설명합니다.
Redis pipelining is a technique for improving performance by issuing multiple commands at once without waiting for the response to each individual command. Pipelining is supported by most Redis clients. This document describes the problem that pipelining is designed to solve and how pipelining works in Redis.

요청/응답 프로토콜 및 왕복 시간
Request/Response protocols and Round-Trip Time (RTT)

Redis는 클라이언트-서버 모델과 소위 요청/응답 프로토콜을 사용하는 TCP 서버입니다. 이는 일반적으로 요청이 다음 단계를 통해 수행됨을 의미합니다.
Redis is a TCP server using the client-server model and what is called a Request/Response protocol.
This means that usually a request is accomplished with the following steps:

- 클라이언트는 서버에 쿼리를 보내고 일반적으로 차단 방식으로 소켓에서 서버 응답을 읽습니다.
  The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
- 서버는 명령을 처리하고 클라이언트에 응답을 다시 보냅니다.
  The server processes the command and sends the response back to the client.

예를 들어 4개의 명령 시퀀스는 다음과 같습니다.
So for instance a four commands sequence is something like this:

```
Client: INCR X
Server: 1
Client: INCR X
Server: 2
Client: INCR X
Server: 3
Client: INCR X
Server: 4
```

클라이언트와 서버는 네트워크 링크를 통해 연결됩니다. 이러한 링크는 매우 빠르거나(루프백 인터페이스) 매우 느릴 수 있습니다 (두 호스트 사이에 많은 홉이 있는 인터넷을 통해 설정된 연결). 네트워크 대기 시간이 무엇이든 패킷이 클라이언트에서 서버로 이동하고 응답을 전달하기 위해 서버에서 클라이언트로 다시 이동하는 데는 시간이 걸립니다.
Clients and Servers are connected via a network link. Such a link can be very fast (a loopback interface) or very slow (a connection established over the Internet with many hops between the two hosts). Whatever the network latency is, it takes time for the packets to travel from the client to the server, and back from the server to the client to carry the reply.

이 시간을 RTT(Round Trip Time)이라고 합니다. 클라이언트가 연속적으로 많은 요청을 수행해야 할 때(예를 들어 동일한 목록에 많은 요소를 추가하거나 많은 키로 데이터베이스를 채우는 경우) 이것이 성능에 어떤 영향을 미칠 수 있는지 쉽게 알 수 있습니다. 예를 들어 RTT 시간이 250밀리초인 경우(인터넷을 통한 링크가 매우 느린 경우) 서버가 초당 100,000개의 요청을 처리할 수 있더라도 초당 최대 4개의 요청을 처리할 수 있습니다.
This time is called RTT (Round Trip Time). It's easy to see how this can affect performance when a client needs to perform many requests in a row (for instance adding many elements to the same list, or populating a database with many keys). For instance if the RTT time is 250 milliseconds (in the case of a very slow link over the Internet), even if the server is able to process 100k requests per second, we'll be able to process at max four requests per second.

사용된 인터페이스가 루프백 인터페이스인 경우 RTT는 훨씬 더 짧으며 일반적으로 1밀리초 미만이지만 연속해서 많은 쓰기를 수행해야 하는 경우에는 이 값도 추가됩니다.
If the interface used is a loopback interface, the RTT is much shorter, typically sub-millisecond, but even this will add up to a lot if you need to perform many writes in a row.

다행히도 이 사용 사례를 개선할 수 있는 방법이 있습니다.
Fortunately there is a way to improve this use case.

레디스 파이프라이닝 Redis Pipelining

클라이언트가 이전 응답을 아직 읽지 않은 경우에도 새 요청을 처리할 수 있도록 요청/응답 서버를 구현할 수 있습니다. 이렇게 하면 응답을 전혀 기다리지 않고 서버에 여러 명령을 보내고 마지막으로 한 단계로 응답을 읽을 수 있습니다.
A Request/Response server can be implemented so that it is able to process new requests even if the client hasn't already read the old responses. This way it is possible to send multiple commands to the server without waiting for the replies at all, and finally read the replies in a single step.

이를 파이프라이닝이라고 하며 수십 년 동안 널리 사용되는 기술입니다. 예를 들어 많은 POP3 프로토콜 구현은 이미 이 기능을 지원하여 서버에서 새 이메일을 다운로드하는 프로세스 속도를 극적으로 향상시킵니다.
This is called pipelining, and is a technique widely in use for many decades. For instance many POP3 protocol implementations already support this feature, dramatically speeding up the process of downloading new emails from the server.

Redis는 초기부터 파이프라이닝을 지원해왔으므로 실행 중인 버전에 관계없이 Redis에서 파이프라이닝을 사용할 수 있습니다. 다음은 원시 netcat 유틸리티를 사용하는 예입니다.
Redis has supported pipelining since its early days, so whatever version you are running, you can use pipelining with Redis. This is an example using the raw netcat utility:

```
$ (printf "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
$ printf "PING\r\nPING\r\nPING\r\n" | nc localhost 6379           -> 괄호() 없어도 가능
$ (printf "auth password\r\nPING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6000  -> auth 추가
+OK
+PONG
+PONG
+PONG
```

이번에는 모든 호출에 대해 RTT 비용을 지불하지 않고 세 가지 명령에 대해 한 번만 지불합니다.
This time we don't pay the cost of RTT for every call, but just once for the three commands.

명확히 말하면 첫 번째 예제의 작업 순서는 파이프라인을 통해 다음과 같습니다.
To be explicit, with pipelining the order of operations of our very first example will be the following:

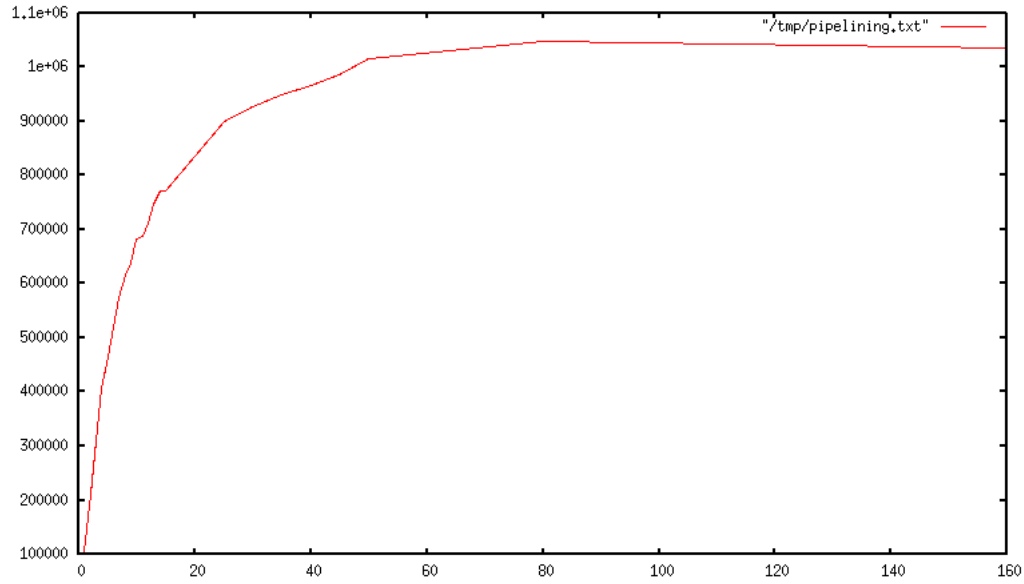
```
Client: INCR X
Client: INCR X
Client: INCR X
Client: INCR X
Server: 1
Server: 2
Server: 3
Server: 4

$ (printf "auth password\r\nINCR X\r\nINCR X\r\nINCR X\r\nINCR X\r\n"; sleep 1)
  | nc localhost 6000
:1
:2
:3
:4
```

중요 참고 사항: 클라이언트가 파이프라이닝을 사용하여 명령을 보내는 동안 서버는 메모리를 사용하여 응답을 대기열에 추가해야 합니다. 따라서 파이프라이닝을 사용하여 많은 명령을 보내야 하는 경우 각각 합리적인 수(예: 10k 명령)를 포함하는 배치로 보내고 응답을 읽은 다음 또 다른 10k 명령을 다시 보내는 것이 좋습니다. 속도는 거의 동일하지만 사용되는 추가 메모리는 기껏해야 이러한 10k 명령에 대한 응답을 대기열에 넣는 데 필요한 양입니다.
IMPORTANT NOTE: While the client sends commands using pipelining, the server will be forced to queue the replies, using memory. So if you need to send a lot of commands with pipelining, it is better to send them as batches each containing a reasonable number, for instance 10k commands, read the replies, and then send another 10k commands again, and so forth. The speed will be nearly the same, but the additional memory used will be at most the amount needed to queue the replies for these 10k commands.

RTT만의 문제가 아닙니다. It's not just a matter of RTT

파이프라이닝은 왕복 시간과 관련된 대기 시간 비용을 줄이는 방법일 뿐만 아니라 실제로 특정 Redis 서버에서 초당 수행할 수 있는 작업 수를 크게 향상시킵니다. 파이프라이닝을 사용하지 않으면 각 명령을 제공하는 것이 데이터 구조에 액세스하고 응답을 생성하는 관점에서는 매우 저렴하지만 소켓 I/O를 수행하는 관점에서는 비용이 매우 많이 들기 때문입니다. 여기에는 read() 및 write() 시스템 호출 호출이 포함됩니다. 이는 사용자 영역에서 커널 영역으로 이동하는 것을 의미합니다. 컨텍스트 전환(context switch)은 속도를 크게 떨어뜨립니다.
Pipelining is not just a way to reduce the latency cost associated with the round trip time, it actually greatly improves the number of operations you can perform per second in a given Redis server. This is because without using pipelining, serving each command is very cheap from the point of view of accessing the data structures and producing the reply, but it is very costly from the point of view of doing the socket I/O. This involves calling the read() and write() syscall, that means going from user land to kernel land. The context switch is a huge speed penalty.

파이프라이닝을 사용하면 일반적으로 단일 read() 시스템 호출로 많은 명령을 읽고 단일 write() 시스템 호출로 여러 응답을 전달합니다. 결과적으로, 초당 수행되는 총 쿼리 수는 처음에는 파이프라인이 길어질수록 거의 선형적으로 증가하며, 이 그림에 표시된 대로 결국 파이프라인 없이 얻은 기준의 10배에 도달합니다.
When pipelining is used, many commands are usually read with a single read() system call, and multiple replies are delivered with a single write() system call. Consequently, the number of total queries performed per second initially increases almost linearly with longer pipelines, and eventually reaches 10 times the baseline obtained without pipelining, as shown in this figure.



실제 코드 예제 A real world code example

다음 벤치마크에서는 파이프라이닝을 지원하는 Redis Ruby 클라이언트를 사용하여 파이프라이닝으로 인한 속도 향상을 테스트합니다.
In the following benchmark we'll use the Redis Ruby client, supporting pipelining, to test the speed improvement due to pipelining:

```
require 'rubygems'
require 'redis'

def bench(descr)
  start = Time.now
  yield
  puts "#{descr} #{Time.now - start} seconds"
end

def without_pipelining
  r = Redis.new
  10_000.times do
    r.ping
  end
end

def with_pipelining
  r = Redis.new
  r.pipelined do
    10_000.times do
      r.ping
    end
  end
end

bench('without pipelining') do
  without_pipelining
end
bench('with pipelining') do
  with_pipelining
end
```

위의 간단한 스크립트를 실행하면 루프백 인터페이스를 통해 실행되는 Mac OS X 시스템에서 다음 그림이 생성됩니다. 여기서 파이프라이닝은 RTT가 이미 매우 낮기 때문에 가장 작은 개선을 제공합니다.
Running the above simple script yields the following figures on my Mac OS X system, running over the loopback interface, where pipelining will provide the smallest improvement as the RTT is already pretty low:

```
without pipelining    1.185238 seconds
with pipelining       0.250783 seconds
```

보시다시피 파이프라이닝을 사용하여 전송을 5배 향상했습니다.
As you can see, using pipelining, we improved the transfer by a factor of five.

Pipelining vs Scripting

Redis 2.6부터 사용 가능한 Redis 스크립팅을 사용하면 서버 측에서 필요한 많은 작업을 수행하는 스크립트를 사용하여 파이프라이닝에 대한 다양한 사용 사례를 보다 효율적으로 처리할 수 있습니다. 스크립팅의 가장 큰 장점은 최소한의 대기 시간으로 데이터를 읽고 쓸 수 있어 읽기, 계산, 쓰기와 같은 작업이 매우 빠르게 수행된다는 것입니다. 클라이언트는 쓰기 명령을 호출하기 전에 읽기 명령의 응답이 필요하므로 이 시나리오에서는 파이프라인이 도움이 되지 않습니다.
Using Redis scripting, available since Redis 2.6, a number of use cases for pipelining can be addressed more efficiently using scripts that perform a lot of the work needed at the server side. A big advantage of scripting is that it is able to both read and write data with minimal latency, making operations like read, compute, write very fast (pipelining can't help in this scenario since the client needs the reply of the read command before it can call the write command).

때로는 애플리케이션이 파이프라인에서 EVAL 또는 EVALSHA 명령을 보내려고 할 수도 있습니다. 이는 전적으로 가능하며 Redis는 SCRIPT LOAD 명령을 통해 이를 명시적으로 지원합니다 (실패 위험 없이 EVALSHA를 호출할 수 있음을 보장함).
Sometimes the application may also want to send EVAL or EVALSHA commands in a pipeline. This is entirely possible and Redis explicitly supports it with the SCRIPT LOAD command (it guarantees that EVALSHA can be called without the risk of failing).

부록: 루프백 인터페이스에서도 사용 중 루프가 느린 이유는 무엇입니까?
Appendix: Why are busy loops slow even on the loopback interface?

이 페이지에서 다룬 모든 배경 지식에도 불구하고 다음과 같은 Redis 벤치마크(의사 코드)가 서버와 클라이언트가 동일한 물리적 시스템에서 실행 중일 때 루프백 인터페이스에서 실행될 때에도 느린 이유가 여전히 궁금할 수 있습니다.
Even with all the background covered in this page, you may still wonder why a Redis benchmark like the following (in pseudo code), is slow even when executed in the loopback interface, when the server and the client are running in the same physical machine:

```
FOR-ONE-SECOND:
  Redis.SET("foo","bar")
END
```

결국 Redis 프로세스와 벤치마크가 모두 동일한 상자에서 실행되는 경우 실제 대기 시간이나 네트워킹이 포함되지 않고 메모리의 메시지를 한 위치에서 다른 위치로 복사하는 것 아닌가요?
After all, if both the Redis process and the benchmark are running in the same box, isn't it just copying messages in memory from one place to another without any actual latency or networking involved?

그 이유는 시스템의 프로세스가 항상 실행되는 것은 아니며 실제로 프로세스를 실행하는 것은 커널 스케줄러이기 때문입니다. 예를 들어 벤치마크 실행이 허용되면 Redis 서버에서 (마지막으로 실행된 명령과 관련된) 응답을 읽고 새 명령을 작성합니다. 명령은 이제 루프백 인터페이스 버퍼에 있지만 서버에서 읽으려면 커널이 서버 프로세스(현재 시스템 호출에서 차단됨)가 실행되도록 예약해야 합니다. 따라서 실용적인 측면에서 루프백 인터페이스에는 커널 스케줄러 작동 방식으로 인해 여전히 네트워크와 유사한 대기 시간이 포함됩니다.
The reason is that processes in a system are not always running, actually it is the kernel scheduler that lets the process run. So, for instance, when the benchmark is allowed to run, it reads the reply from the Redis server (related to the last command executed), and writes a new command. The command is now in the loopback interface buffer, but in order to be read by the server, the kernel should schedule the server process (currently blocked in a system call) to run, and so forth. So in practical terms the loopback interface still involves network-like latency, because of how the kernel scheduler works.

기본적으로 비지 루프 벤치마크는 네트워크로 연결된 서버의 성능을 측정할 때 수행할 수 있는 가장 어리석은 작업입니다. 현명한 것은 이런 식으로 벤치마킹을 피하는 것입니다.

Basically a busy loop benchmark is the silliest thing that can be done when metering performances on a networked server. The wise thing is just avoiding benchmarking in this way.

2. Spring Pipelining (Spring Data Redis 문서)

Redis는 응답을 기다리지 않고 여러 명령을 서버에 보낸 다음 단일 단계로 응답을 읽는 파이프라이닝을 지원합니다 . 파이프라이닝은 동일한 목록에 많은 요소를 추가하는 등 여러 명령을 연속으로 보내야 할 때 성능을 향상시킬 수 있습니다.
Spring Data Redis는 RedisTemplate 파이프라인에서 명령을 실행하기 위한 여러 가지 방법을 제공합니다. 파이프라인 작업의 결과에 관심이 없다면, 파이프라인 인수에 true를 전달하여 표준 execute 메서드를 사용할 수 있습니다.
다음 예제와 같이 executePipelined 메서드는 제공된 RedisCallback 또는 SessionCallback를 파이프라인에서 실행하고 결과를 반환합니다.

Redis provides support for pipelining, which involves sending multiple commands to the server without waiting for the replies and then reading the replies in a single step. Pipelining can improve performance when you need to send several commands in a row, such as adding many elements to the same List.
Spring Data Redis provides several RedisTemplate methods for running commands in a pipeline. If you do not care about the results of the pipelined operations, you can use the standard execute method, passing true for the pipeline argument. The executePipelined methods run the provided RedisCallback or SessionCallback in a pipeline and return the results, as shown in the following example:

```
//pop a specified number of items from a queue
List<Object> results = stringRedisTemplate.executePipelined(
  new RedisCallback<Object>() {
    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      StringRedisConnection stringRedisConn = (StringRedisConnection)connection;
      for(int i=0; i< batchSize; i++) {
        stringRedisConn.rPop("myqueue");
      }
    return null;
  }
});
```

앞의 예에서는 파이프라인의 대기열에서 항목의 대량 RPOP을 실행합니다. 여기에는 results List에는 팝업된 항목이 모두 포함되어 있습니다. RedisTemplate 해당 값, 해시 키 및 해시 값 직렬 변환기를 사용하여 모든 결과를 반환하기 전에 역직렬화하므로, 예제에서 반환된 항목은 문자열입니다. executePipelined 파이프라인 결과에 대해 사용자 지정 직렬 변환기를 전달할 수 있는 추가 메서드가 있습니다.
파이프라인 명령의 결과를 반환하기 위해 이 값이 삭제되므로 RedisCallback에서 반환된 값은 null 이어야 합니다.

The preceding example runs a bulk right pop of items from a queue in a pipeline. The results List contains all of the popped items. RedisTemplate uses its value, hash key, and hash value serializers to deserialize all results before returning, so the returned items in the preceding example are Strings. There are additional executePipelined methods that let you pass a custom serializer for pipelined results.
Note that the value returned from the RedisCallback is required to be null, as this value is discarded in favor of returning the results of the pipelined commands.

> Lettuce 드라이버는 명령이 나타날 때 플러시하거나 버퍼링하거나 연결이 닫힐 때 보낼 수 있는 세분화된 플러시 제어를 지원합니다.
> LettuceConnectionFactory factory = // ...
> // 로컬로 버퍼링하고 세 번째 명령마다 플러시합니다.
> factory.setPipeliningFlushPolicy(PipeliningFlushPolicy.buffered(3));

3. Java Pipelining Source

Java Spring Framework를 사용한 레디스 파이프라인(Pipelining) 명령 사용법입니다.

파이프라인 실행(stringRedisTemplate.executePipelined)은 새 연결을 맺어서 처리하고 종료(close)합니다.
예제 1)은 SET 명령을 100번 실행합니다.
예제 2)는 RPOP 명령을 100번 실행합니다.
  예제 2)를 실행하기 전에 List 예제 9) http://localhost:8080/listinput/mylist9 를 먼저 실행해서 데이터를 입력하세요.
100개의 명령이 한번에 보내고, 결과를 한번에 받습니다.
대량(bulk/batch) 명령 처리에 사용하면 성능이 좋습니다.

Redis08_Pipeline.java

```
package com.redisgate.redis;

import lombok.extern.slf4j.Slf4j;
import org.springframework.dao.DataAccessException;
import org.springframework.data.redis.connection.RedisConnection;
import org.springframework.data.redis.connection.StringRedisConnection;
import org.springframework.data.redis.core.RedisCallback;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@Slf4j
public class Redis08_Pipeline {

   private final StringRedisTemplate stringRedisTemplate;

   public Redis08_Pipeline(StringRedisTemplate stringRedisTemplate) {
      this.stringRedisTemplate = stringRedisTemplate;
   }
```

```
  // 예제 1) SET 100번 입력
  // 새 연결을 맺어서 처리하고 종료(close)한다.
  // http://localhost:8080/pipe/100
  @GetMapping("/pipeset/{count}")
  public String pipeset(@PathVariable("count") int count) {
     String msg = "예제 1) Pipeline(SET) -> ";
     List<Object> results = stringRedisTemplate.executePipelined(new RedisCallback<Object>() {
        public Object doInRedis(RedisConnection connection) throws DataAccessException {
           StringRedisConnection stringRedisConn = (StringRedisConnection)connection;
           for(int i=0; i< count; i++) {
              stringRedisConn.set("keyA"+String.format("-%05d",i),"value-"+String.format("%05d",i));
           }
           return null;
        }
     });
     System.out.println(msg+results.size());
     results.forEach(System.out::println);
     return msg+results.size();
  }
```

```
  // 예제 2) RPOP 100번 실행
  // 새 연결을 맺어서 처리하고 종료(close)한다.
  // List 예제 9) 에서 입력한 데이터를 사용한다.
  // http://localhost:8080/piperpop/100

  @GetMapping("/piperpop/{count}")
  public String piperpop(@PathVariable("count") int count) {
     String msg = "예제 2) Pipeline(RPOP) -> ";
     List<Object> results = stringRedisTemplate.executePipelined(new RedisCallback<Object>() {
        public Object doInRedis(RedisConnection connection) throws DataAccessException {
           StringRedisConnection stringRedisConn = (StringRedisConnection)connection;
           for(int i=0; i< count; i++) {
              stringRedisConn.rPop("mylist9");
           }
           return null;
        }
     });
     System.out.println(msg+results.size());
     results.forEach(System.out::println);
     return msg+results.size();
  }
}
```

<< Common Keys Pipelining Pub/Sub >>

redisgate@gmail.com 02.503.2235
서울시 강남구 강남대로 342 역삼빌딩 5층 (역삼동) 우 06242