目　录

致谢

当前文档 《Redisson 官方文档中文翻译》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-11。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/redisson-doc-cn

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

redisson-doc-cn

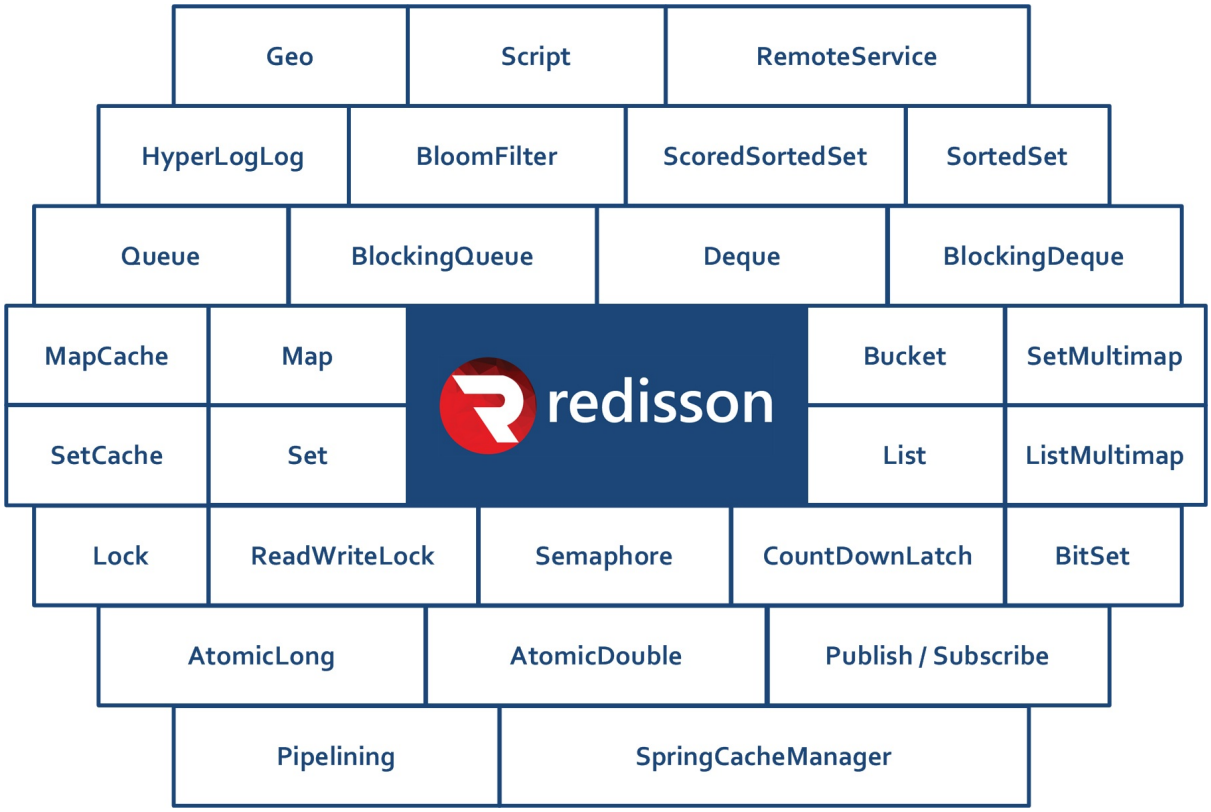
Redisson 官方文档中文翻译。

来源(书栈小编注)

> *https://github.com/liulongbiao/redisson-doc-cn*

概览

概览



Redisson 不只是一个 Java Redis 客户端，
它是一个以内存 Redis 服务器作为后端的处理 Java 对象
(如 `java.util.List`， `java.util.Map`， `java.util.Set`，
`java.util.concurrent.locks.Lock` 等)
的一个框架。
它也提供了一些高级服务，如
RemoteService 和
SpringCacheManager。
Redisson 的目标是提供使用 Redis 的更方便和容易的方式。

若你已经使用过其他 Java Redis 客户端，在迁移到 Redisson 过程中
Redis命令映射 列表会很有用。

每个 Redis 服务器实例最多可处理 1TB 内存。

基于 Netty 框架。
Redis 2.8+ 和 JDK 1.6+ 兼容性。

配置

配置

TOC

编程式配置

编程式配置通过 `Config` 对象实例来执行。如：

```
Config config = new Config();
config.setUseLinuxNativeEpoll(true);
config.useClusterServers()
        .addNodeAddress("127.0.0.1:7181");
```

声明式配置

Redisson 配置可从 JSON 或 YAML 格式的文件中加载。

要从 JSON 读取配置，可使用 `Config.fromJSON` 方法指向配置源来完成：

```
Config config = Config.fromJSON(new File("config-file.json"));
RedissonClient redisson = Redisson.create(config);
```

要将配置写出为 JSON ，可使用 `Config.toJSON` 方法：

```
Config config = new Config();
// ... many settings are set here
String jsonFormat = config.toJSON();
```

要从 YAML 读取配置，可使用 `Config.fromYAML` 方法指向配置源来完成：

```
Config config = Config.fromYAML(new File("config-file.yaml"));
RedissonClient redisson = Redisson.create(config);
```

要将配置写出为 YAML ，可使用 `Config.toYAML` 方法：

```
Config config = new Config();
// ... many settings are set here
String yamlFormat = config.toYAML();
```

通用设置

以下设置属于 `org.redisson.Config` 对象，且对所有模式都是通用的：

codec

默认值： `org.redisson.codec.JsonJacksonCodec`

Redis 数据解编码器。用于读写 Redis 数据。支持多种实现：

Codec 类名	描述
`org.redisson.codec.JsonJacksonCodec`	Jackson JSON codec

`org.redisson.codec.CborJacksonCodec`	CBOR 二进制 json codec
`org.redisson.codec.MsgPackJacksonCodec`	MsgPack 二进制 json codec
`org.redisson.codec.KryoCodec`	Kryo 二进制 codec
`org.redisson.codec.SerializationCodec`	JDK 序列化 codec
`org.redisson.codec.FstCodec`	FST 10 倍速且 100% JDK 序列化兼容的 codec
`org.redisson.codec.LZ4Codec`	LZ4 压缩 codec
`org.redisson.codec.SnappyCodec`	Snappy 压缩 codec
`org.redisson.client.codec.StringCodec`	简单 String codec
`org.redisson.client.codec.LongCodec`	简单 Long codec

threads

默认值： `current_processors_amount * 2`

由所有 redis 节点客户端所共享的线程数量。

eventLoopGroup

使用外部的 `EventLoopGroup`。EventLoopGroup 通过自己的线程来处理所有和 Redis 服务器相连的 Netty 链接。
每个 Redisson 客户端会默认创建一个自己的 EventLoopGroup。因此若相同的 JVM 中存在多个 Redisson 实例，它会在它们之间共享一个 EventLoopGroup。

只允许使用 `io.netty.channel.epoll.EpollEventLoopGroup` 或 `io.netty.channel.nio.NioEventLoopGroup`。

useLinuxNativeEpoll

默认值： `false`

当服务器绑定到 loopback 接口时激活一个 unix socket。
同样也用于激活 epoll 协议。

`netty-transport-native-epoll` 类库需包含在 classpath 中。

集群模式" class="reference-link"> 集群模式

编程式配置示例：

```
1. Config config = new Config();
2. config.useClusterServers()
3.     .setScanInterval(2000) // cluster state scan interval in
   milliseconds
4.     .addNodeAddress("127.0.0.1:7000", "127.0.0.1:7001")
5.     .addNodeAddress("127.0.0.1:7002");
6.
7. RedissonClient redisson = Redisson.create(config);
```

集群设置

有关 Redis 服务器集群配置的文档在 这里。
最小的集群配置需要至少三个主节点。
集群连接模式可通过以下代码激活：

```
1. ClusterServersConfig clusterConfig = config.useClusterServers();
```

`ClusterServersConfig` 设置项如下：

addNodeAddress

以 `host:port` 格式添加 Redis 集群节点地址。可以一次性添加多个节点。

scanInterval

默认值： `1000`

以毫秒为单位的 Redis 集群扫描间隔。

readFromSlaves

默认值: `true`

读操作是否可使用集群从节点。

loadBalancer

默认值: `org.redisson.connection.balancer.RoundRobinLoadBalancer`

其它实现有:

`org.redisson.connection.balancer.WeightedRoundRobinBalancer`,
`redisson.connection.balancer.RandomLoadBalancer`

多个 Redis 从服务器间的连接负载均衡器。

slaveSubscriptionConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲订阅 (pub/sub) 连接量。

slaveSubscriptionConnectionPoolSize

默认值: `25`

对 每个 从节点的 Redis ‘从’节点最大订阅 (pub/sub) 连接池大小。

slaveConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲连接量。

slaveConnectionPoolSize

默认值: `100`

对 每个 从节点的 Redis ‘从’节点最大连接池大小。

masterConnectionMinimumIdleSize

默认值: `5`

对 每个 从节点的 Redis ‘主’节点最小空闲连接量。

masterConnectionPoolSize

默认值: `100`

Redis ‘主’节点最大连接池大小。

idleConnectionTimeout

默认值: `10000`

若池化连接在某段 `timeout` 时间内没有被使用且当前连接量超过最小空闲连接池时，
它将会被关闭并从池中移除。其值的单位是毫秒。

connectTimeout

默认值: `1000`

连接到任何 Redis 服务器的超时时间。

timeout

默认值： `1000`

Redis 服务器响应的超时时间。从 Redis 命令被成功发送时开始计算。其值的单位是毫秒。

retryAttempts

默认值： `3`

若 Redis 命令在超过 `retryAttempts` 次不能发送被 Redis 服务器，则将抛出一个错误。
但若成功发送，则将开始 `timeout`。

retryInterval

默认值： `1000`

发送 Redis 命令重试的时间间隔。其值的单位是毫秒。

reconnectionTimeout

默认值： `3000`

Redis 服务器重连尝试的超时时间。在每次这种超时事件发生时，Redisson 会尝试连接到失联的 Redis 服务器。
其值的单位是毫秒。

failedAttempts

默认值： `3`

当任何 Redis 命令的连续的未成功执行尝试到达 `failedAttempts` 时，
这个 Redis 服务器将被从一个内部的可用从节点列表中移除。

database

默认值: `0`

针对 Redis 连接的数据库索引。

password

默认值: `null`

Redis 服务器授权的密码。

subscriptionsPerConnection

默认值: `5`

每个 Redis 连接上的订阅的限制

clientName

默认值: `null`

客户端连接的名称。

集群 JSON 和 YAML 配置格式

以下是 JSON 格式的集群配置示例。
所有的属性名称都匹配 `ClusterServersConfig` 和 `Config` 对象的属性名称。

```
{
   "clusterServersConfig":{
      "idleConnectionTimeout":10000,
      "pingTimeout":1000,
      "connectTimeout":1000,
      "timeout":1000,
```

```
        "retryAttempts":3,
        "retryInterval":1000,
        "reconnectionTimeout":3000,
        "failedAttempts":3,
        "password":null,
        "subscriptionsPerConnection":5,
        "clientName":null,
        "loadBalancer":{
   "class":"org.redisson.connection.balancer.RoundRobinLoadBalancer"
        },
        "slaveSubscriptionConnectionMinimumIdleSize":1,
        "slaveSubscriptionConnectionPoolSize":25,
        "slaveConnectionMinimumIdleSize":5,
        "slaveConnectionPoolSize":100,
        "masterConnectionMinimumIdleSize":5,
        "masterConnectionPoolSize":100,
        "readMode":"SLAVE",
        "nodeAddresses":[
           "//127.0.0.1:7004",
           "//127.0.0.1:7001",
           "//127.0.0.1:7000"
        ],
        "scanInterval":1000
    },
    "threads":0,
    "codec":null,
    "useLinuxNativeEpoll":false,
    "eventLoopGroup":null
}
```

以下是 YAML 格式的集群配置示例。

所有的属性名称都匹配 `ClusterServersConfig` 和 `Config` 对象的属性名称。

```
---
clusterServersConfig:
```

```
  idleConnectionTimeout: 10000
  pingTimeout: 1000
  connectTimeout: 1000
  timeout: 1000
  retryAttempts: 3
  retryInterval: 1000
  reconnectionTimeout: 3000
  failedAttempts: 3
  password: null
  subscriptionsPerConnection: 5
  clientName: null
  loadBalancer: !
<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
  slaveSubscriptionConnectionMinimumIdleSize: 1
  slaveSubscriptionConnectionPoolSize: 25
  slaveConnectionMinimumIdleSize: 5
  slaveConnectionPoolSize: 100
  masterConnectionMinimumIdleSize: 5
  masterConnectionPoolSize: 100
  readMode: "SLAVE"
  nodeAddresses:
  - "//127.0.0.1:7004"
  - "//127.0.0.1:7001"
  - "//127.0.0.1:7000"
  scanInterval: 1000
threads: 0
codec: !<org.redisson.codec.JsonJacksonCodec> {}
useLinuxNativeEpoll: false
eventLoopGroup: null
```

Elasticache 模式

编程式配置示例:

```
Config config = new Config();
config.useElasticacheServers()
    .setScanInterval(2000) // 主节点变更扫描间隔
```

```
4.      .addNodeAddress("127.0.0.1:7000", "127.0.0.1:7001")
5.      .addNodeAddress("127.0.0.1:7002");
6.
7.  RedissonClient redisson = Redisson.create(config);
```

Elasticache 设置

有关 AWS Elasticache Redis 服务器配置的文档在这里。

Elasticache 连接模式可由以下代码激活:

```
1.  ElasticacheServersConfig clusterConfig =
    config.useElasticacheServers();
```

`ElasticacheServersConfig` 设置项如下:

addNodeAddress

以 `host:port` 格式添加 Redis 集群节点地址。可以一次添加多个节点。
所有的节点(主节点和从节点)必须在配置时提供。

scanInterval

默认值: `1000`

以毫秒为单位的 Elasticache 节点扫描间隔。

loadBalancer

默认值: `org.redisson.connection.balancer.RoundRobinLoadBalancer`

其它实现:

`org.redisson.connection.balancer.WeightedRoundRobinBalancer`,

`org.redisson.connection.balancer.RandomLoadBalancer`

多个 Redis 从服务器间的连接负载均衡器

slaveSubscriptionConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲订阅 (pub/sub) 连接量。

slaveSubscriptionConnectionPoolSize

默认值: `25`

对 每个 从节点的 Redis ‘从’节点最大订阅 (pub/sub) 连接池大小。

slaveConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲连接量。

slaveConnectionPoolSize

默认值: `100`

对 每个 从节点的 Redis ‘从’节点最大连接池大小。

masterConnectionMinimumIdleSize

默认值: `5`

对 每个 从节点的 Redis ‘主’节点最小空闲连接量。

masterConnectionPoolSize

默认值: `100`

Redis ‘主’节点最大连接池大小。

idleConnectionTimeout

默认值: `10000`

若池化连接在某段 `timeout` 时间内没有被使用且当前连接量超过最小空闲连接池时，
它将会被关闭并从池中移除。其值的单位是毫秒。

connectTimeout

默认值: `1000`

连接到任何 Redis 服务器的超时时间。

timeout

默认值: `1000`

Redis 服务器响应的超时时间。从 Redis 命令被成功发送时开始计算。其值的单位是毫秒。

retryAttempts

默认值: `3`

若 Redis 命令在超过 `retryAttempts` 次不能发送被 Redis 服务器，则将抛出一个错误。
但若成功发送，则将开始 `timeout` 。

retryInterval

默认值: `1000`

发送 Redis 命令重试的时间间隔。其值的单位是毫秒。

reconnectionTimeout

默认值: `3000`

Redis 服务器重连尝试的超时时间。在每次这种超时事件发生时，
Redisson 会尝试连接到失联的 Redis 服务器。
其值的单位是毫秒。

failedAttempts

默认值: `3`

当任何 Redis 命令的连续的未成功执行尝试到达 `failedAttempts` 时，
这个 Redis 服务器将被从一个内部的可用从节点列表中移除。

database

默认值: `0`

针对 Redis 连接的数据库索引。

password

默认值: `null`

Redis 服务器授权的密码。

subscriptionsPerConnection

默认值： `5`

每个 Redis 连接上的订阅的限制

clientName

默认值： `null`

客户端连接的名称。

Elasticache JSON 和 YAML 配置格式

以下是 JSON 格式的 Elasticache 配置示例。
所有的属性名称都匹配 `ElasticacheServersConfig` 和 `Config` 对象的属性名称。

```
{
   "elasticacheServersConfig":{
      "idleConnectionTimeout":10000,
      "pingTimeout":1000,
      "connectTimeout":1000,
      "timeout":1000,
      "retryAttempts":3,
      "retryInterval":1000,
      "reconnectionTimeout":3000,
      "failedAttempts":3,
      "password":null,
      "subscriptionsPerConnection":5,
      "clientName":null,
      "loadBalancer":{
         "class":"org.redisson.connection.balancer.RoundRobinLoadBalancer"
      },
      "slaveSubscriptionConnectionMinimumIdleSize":1,
      "slaveSubscriptionConnectionPoolSize":25,
      "slaveConnectionMinimumIdleSize":5,
```

```
      "slaveConnectionPoolSize":100,
      "masterConnectionMinimumIdleSize":5,
      "masterConnectionPoolSize":100,
      "readMode":"SLAVE",
      "nodeAddresses":[
         "//127.0.0.1:2812",
         "//127.0.0.1:2815",
         "//127.0.0.1:2813"
      ],
      "scanInterval":1000,
      "database":0
   },
   "threads":0,
   "codec":null,
   "useLinuxNativeEpoll":false,
   "eventLoopGroup":null
}
```

以下是 YAML 格式的 Elasticache 配置示例。所有的属性名称都匹配 `ElasticacheServersConfig` 和 `Config` 对象的属性名称。

```
---
elasticacheServersConfig:
  idleConnectionTimeout: 10000
  pingTimeout: 1000
  connectTimeout: 1000
  timeout: 1000
  retryAttempts: 3
  retryInterval: 1000
  reconnectionTimeout: 3000
  failedAttempts: 3
  password: null
  subscriptionsPerConnection: 5
  clientName: null
  loadBalancer: !
<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
```

```
15.    slaveSubscriptionConnectionMinimumIdleSize: 1
16.    slaveSubscriptionConnectionPoolSize: 25
17.    slaveConnectionMinimumIdleSize: 5
18.    slaveConnectionPoolSize: 100
19.    masterConnectionMinimumIdleSize: 5
20.    masterConnectionPoolSize: 100
21.    readMode: "SLAVE"
22.    nodeAddresses:
23.    - "//127.0.0.1:2812"
24.    - "//127.0.0.1:2815"
25.    - "//127.0.0.1:2813"
26.    scanInterval: 1000
27.    database: 0
28.  threads: 0
29.  codec: !<org.redisson.codec.JsonJacksonCodec> {}
30.  useLinuxNativeEpoll: false
31.  eventLoopGroup: null
```

单实例模式

编程式配置示例：

```
1.  // connects to 127.0.0.1:6379 by default
2.  RedissonClient redisson = Redisson.create();
3.
4.  Config config = new Config();
5.  config.useSingleServer().setAddress("myredisserver:6379");
6.  RedissonClient redisson = Redisson.create(config);
```

单实例设置

有关 Redis 单服务器配置的文档在
这里。
单服务器连接模式可通过以下代码激活：

```
1.  SingleServerConfig clusterConfig = config.useSingleServer();
```

`SingleServerConfig` 设置项如下:

address

`host:port` 格式的 Redis 服务器地址。

subscriptionConnectionMinimumIdleSize

默认值: `1`

最小空闲Redis订阅(pub/sub)连接量

subscriptionConnectionPoolSize

默认值: `25`

最大Redis订阅 (pub/sub) 连接池大小。

connectionMinimumIdleSize

默认值: `5`

最小Redis空闲连接量。

connectionPoolSize

默认值: `100`

最大Redis连接池大小。

dnsMonitoring

默认值: `false`

若为 `true`，服务器地址将监控 DNS 中的变更

dnsMonitoringInterval

默认值: `5000`

DNS 变更监控间隔

idleConnectionTimeout

默认值: `10000`

若池化连接在某段 `timeout` 时间内没有被使用且当前连接量超过最小空闲连接池时，
它将会被关闭并从池中移除。其值的单位是毫秒。

connectTimeout

默认值: `1000`

连接到任何 Redis 服务器的超时时间。

timeout

默认值: `1000`

Redis 服务器响应的超时时间。从 Redis 命令被成功发送时开始计算。其值的单位是毫秒。

retryAttempts

默认值: `3`

若 Redis 命令在超过 `retryAttempts` 次不能发送被 Redis 服务器，则将抛出一个错误。

但若成功发送，则将开始 `timeout` 。

retryInterval

默认值: `1000`

发送 Redis 命令重试的时间间隔。其值的单位是毫秒。

reconnectionTimeout

默认值: `3000`

Redis 服务器重连尝试的超时时间。在每次这种超时事件发生时，
Redisson 会尝试连接到失联的 Redis 服务器。
其值的单位是毫秒。

failedAttempts

默认值: `3`

当任何 Redis 命令的连续的未成功执行尝试到达 `failedAttempts` 时，
这个 Redis 服务器将被从一个内部的可用从节点列表中移除。

database

默认值: `0`

针对 Redis 连接的数据库索引。

password

默认值: `null`

Redis 服务器授权的密码。

subscriptionsPerConnection

默认值: `5`

每个 Redis 连接上的订阅的限制

clientName

默认值: `null`

客户端连接的名称。

单实例 JSON 和 YAML 配置格式

以下是 JSON 格式的单实例配置示例。
所有属性名称都匹配 `SingleServerConfig` 和 `Config` 对象的属性名称。

```
{
   "singleServerConfig":{
      "idleConnectionTimeout":10000,
      "pingTimeout":1000,
      "connectTimeout":1000,
      "timeout":1000,
      "retryAttempts":3,
      "retryInterval":1000,
      "reconnectionTimeout":3000,
      "failedAttempts":3,
      "password":null,
      "subscriptionsPerConnection":5,
      "clientName":null,
      "address":[
         "//127.0.0.1:6379"
      ],
      "subscriptionConnectionMinimumIdleSize":1,
      "subscriptionConnectionPoolSize":25,
      "connectionMinimumIdleSize":5,
```

```
        "connectionPoolSize":100,
        "database":0,
        "dnsMonitoring":false,
        "dnsMonitoringInterval":5000
    },
    "threads":0,
    "codec":null,
    "useLinuxNativeEpoll":false,
    "eventLoopGroup":null
}
```

以下是 YAML 格式的单实例配置示例。
所有属性名称都匹配 `SingleServerConfig` 和 `Config` 对象的属性名称。

```
---
singleServerConfig:
  idleConnectionTimeout: 10000
  pingTimeout: 1000
  connectTimeout: 1000
  timeout: 1000
  retryAttempts: 3
  retryInterval: 1000
  reconnectionTimeout: 3000
  failedAttempts: 3
  password: null
  subscriptionsPerConnection: 5
  clientName: null
  address:
  - "//127.0.0.1:6379"
  subscriptionConnectionMinimumIdleSize: 1
  subscriptionConnectionPoolSize: 25
  connectionMinimumIdleSize: 5
  connectionPoolSize: 100
  database: 0
  dnsMonitoring: false
  dnsMonitoringInterval: 5000
```

```
threads: 0
codec: !<org.redisson.codec.JsonJacksonCodec> {}
useLinuxNativeEpoll: false
eventLoopGroup: null
```

Sentinel 模式

编程式配置示例:

```
Config config = new Config();
config.useSentinelServers()
    .setMasterName("mymaster")
    .addSentinelAddress("127.0.0.1:26389", "127.0.0.1:26379")
    .addSentinelAddress("127.0.0.1:26319");

RedissonClient redisson = Redisson.create(config);
```

Sentinel 设置

有关 Redis 服务器 sentinel 配置的文档在
这里。
集群连接模式可由以下代码激活:

```
SentinelServersConfig clusterConfig = config.useSentinelServers();
```

`SentinelServersConfig` 设置项如下:

masterName

由 Redis Sentinel 服务器和主节点变更监控任务所使用的主服务器名称。

addSentinelAddress

添加 `host:port` 格式的 Redis Sentinel 节点地址。可一次性添加多个节点。

loadBalancer

默认值: `org.redisson.connection.balancer.RoundRobinLoadBalancer`

其它实现有:

`org.redisson.connection.balancer.WeightedRoundRobinBalancer`, `redisson.connection.balancer.RandomLoadBalancer`

多个 Redis 从服务器间的连接负载均衡器。

slaveSubscriptionConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲订阅 (pub/sub) 连接量。

slaveSubscriptionConnectionPoolSize

默认值: `25`

对 每个 从节点的 Redis ‘从’节点最大订阅 (pub/sub) 连接池大小。

slaveConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲连接量。

slaveConnectionPoolSize

默认值: `100`

对 每个 从节点的 Redis '从'节点最大连接池大小。

masterConnectionMinimumIdleSize

默认值: `5`

对 每个 从节点的 Redis '主'节点最小空闲连接量。

masterConnectionPoolSize

默认值: `100`

Redis '主'节点最大连接池大小。

idleConnectionTimeout

默认值: `10000`

若池化连接在某段 `timeout` 时间内没有被使用且当前连接量超过最小空闲连接池时,
它将会被关闭并从池中移除。其值的单位是毫秒。

connectTimeout

默认值: `1000`

连接到任何 Redis 服务器的超时时间。

timeout

默认值: `1000`

Redis 服务器响应的超时时间。从 Redis 命令被成功发送时开始计算。其值的单位是毫秒。

retryAttempts

默认值： `3`

若 Redis 命令在超过 `retryAttempts` 次不能发送被 Redis 服务器，则将抛出一个错误。
但若成功发送，则将开始 `timeout`。

retryInterval

默认值： `1000`

发送 Redis 命令重试的时间间隔。其值的单位是毫秒。

reconnectionTimeout

默认值： `3000`

Redis 服务器重连尝试的超时时间。在每次这种超时事件发生时，Redisson 会尝试连接到失联的 Redis 服务器。
其值的单位是毫秒。

failedAttempts

默认值： `3`

当任何 Redis 命令的连续的未成功执行尝试到达 `failedAttempts` 时，
这个 Redis 服务器将被从一个内部的可用从节点列表中移除。

database

默认值： `0`

针对 Redis 连接的数据库索引。

password

默认值： `null`

Redis 服务器授权的密码。

subscriptionsPerConnection

默认值： `5`

每个 Redis 连接上的订阅的限制

clientName

默认值： `null`

客户端连接的名称。

Sentinel JSON 和 YAML 配置格式

以下是 JSON 格式的 Sentinel 配置示例。
所有属性名称都匹配 `SentinelServerConfig` 和 `Config` 对象的属性名称。

```
{
   "sentinelServersConfig":{
      "idleConnectionTimeout":10000,
      "pingTimeout":1000,
      "connectTimeout":1000,
      "timeout":1000,
      "retryAttempts":3,
      "retryInterval":1000,
      "reconnectionTimeout":3000,
      "failedAttempts":3,
      "password":null,
      "subscriptionsPerConnection":5,
      "clientName":null,
      "loadBalancer":{
```

```
15.
    "class":"org.redisson.connection.balancer.RoundRobinLoadBalancer"
16.       },
17.       "slaveSubscriptionConnectionMinimumIdleSize":1,
18.       "slaveSubscriptionConnectionPoolSize":25,
19.       "slaveConnectionMinimumIdleSize":5,
20.       "slaveConnectionPoolSize":100,
21.       "masterConnectionMinimumIdleSize":5,
22.       "masterConnectionPoolSize":100,
23.       "readMode":"SLAVE",
24.       "sentinelAddresses":[
25.          "//127.0.0.1:26379",
26.          "//127.0.0.1:26389"
27.       ],
28.       "masterName":"mymaster",
29.       "database":0
30.    },
31.    "threads":0,
32.    "codec":null,
33.    "useLinuxNativeEpoll":false,
34.    "eventLoopGroup":null
35. }
```

以下是 YAML 格式的 Sentinel 配置示例。

所有属性名称都匹配 `SentinelServerConfig` 和 `Config` 对象的属性名称。

```
1.  ---
2.  sentinelServersConfig:
3.    idleConnectionTimeout: 10000
4.    pingTimeout: 1000
5.    connectTimeout: 1000
6.    timeout: 1000
7.    retryAttempts: 3
8.    retryInterval: 1000
9.    reconnectionTimeout: 3000
10.   failedAttempts: 3
```

```
  password: null
  subscriptionsPerConnection: 5
  clientName: null
  loadBalancer: !
<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
  slaveSubscriptionConnectionMinimumIdleSize: 1
  slaveSubscriptionConnectionPoolSize: 25
  slaveConnectionMinimumIdleSize: 5
  slaveConnectionPoolSize: 100
  masterConnectionMinimumIdleSize: 5
  masterConnectionPoolSize: 100
  readMode: "SLAVE"
  sentinelAddresses:
  - "//127.0.0.1:26379"
  - "//127.0.0.1:26389"
  masterName: "mymaster"
  database: 0
threads: 0
codec: !<org.redisson.codec.JsonJacksonCodec> {}
useLinuxNativeEpoll: false
eventLoopGroup: null
```

主从模式

编程式配置示例：

```
Config config = new Config();
config.useMasterSlaveServers()
    .setMasterAddress("127.0.0.1:6379")
    .addSlaveAddress("127.0.0.1:6389", "127.0.0.1:6332",
"127.0.0.1:6419")
    .addSlaveAddress("127.0.0.1:6399");

RedissonClient redisson = Redisson.create(config);
```

主从设置

有关 Redis 服务器主/从配置的文档在
这里。
主从连接模式可由以下代码激活:

```
1.  MasterSlaveServersConfig clusterConfig =
    config.useMasterSlaveServers();
```

`MasterSlaveServersConfig` 设置项如下:

masterAddress

`host:port` 格式的 Redis 主节点地址

addSlaveAddress

添加 `host:port` 格式的 Redis 从节点地址。可一次添加多个节点。

loadBalancer

默认值: `org.redisson.connection.balancer.RoundRobinLoadBalancer`

其它实现有:

`org.redisson.connection.balancer.WeightedRoundRobinBalancer`,
`redisson.connection.balancer.RandomLoadBalancer`

多个 Redis 从服务器间的连接负载均衡器。

slaveSubscriptionConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis '从'节点最小空闲订阅 (pub/sub) 连接量。

slaveSubscriptionConnectionPoolSize

默认值: `25`

对 每个 从节点的 Redis ‘从’节点最大订阅 (pub/sub) 连接池大小。

slaveConnectionMinimumIdleSize

默认值: `1`

对 每个 从节点的 Redis ‘从’节点最小空闲连接量。

slaveConnectionPoolSize

默认值: `100`

对 每个 从节点的 Redis ‘从’节点最大连接池大小。

masterConnectionMinimumIdleSize

默认值: `5`

对 每个 从节点的 Redis ‘主’节点最小空闲连接量。

masterConnectionPoolSize

默认值: `100`

Redis ‘主’节点最大连接池大小。

idleConnectionTimeout

默认值: `10000`

若池化连接在某段 `timeout` 时间内没有被使用且当前连接量超过最

小空闲连接池时，
它将会被关闭并从池中移除。其值的单位是毫秒。

connectTimeout

默认值: `1000`

连接到任何 Redis 服务器的超时时间。

timeout

默认值: `1000`

Redis 服务器响应的超时时间。从 Redis 命令被成功发送时开始计算。其值的单位是毫秒。

retryAttempts

默认值: `3`

若 Redis 命令在超过 `retryAttempts` 次不能发送被 Redis 服务器，则将抛出一个错误。
但若成功发送，则将开始 `timeout` 。

retryInterval

默认值: `1000`

发送 Redis 命令重试的时间间隔。其值的单位是毫秒。

reconnectionTimeout

默认值: `3000`

Redis 服务器重连尝试的超时时间。在每次这种超时事件发生时，

Redisson 会尝试连接到失联的 Redis 服务器。
其值的单位是毫秒。

failedAttempts

默认值: `3`

当任何 Redis 命令的连续的未成功执行尝试到达 `failedAttempts` 时,
这个 Redis 服务器将被从一个内部的可用从节点列表中移除。

database

默认值: `0`

针对 Redis 连接的数据库索引。

password

默认值: `null`

Redis 服务器授权的密码。

subscriptionsPerConnection

默认值: `5`

每个 Redis 连接上的订阅的限制

clientName

默认值: `null`

客户端连接的名称。

主从 JSON 和 YAML 配置格式

以下是 JSON 格式的主从配置示例。
所有属性名称都匹配 `MasterSlaveServersConfig` 和 `Config` 对象的属性名称。

```
{
   "masterSlaveServersConfig":{
      "idleConnectionTimeout":10000,
      "pingTimeout":1000,
      "connectTimeout":1000,
      "timeout":1000,
      "retryAttempts":3,
      "retryInterval":1000,
      "reconnectionTimeout":3000,
      "failedAttempts":3,
      "password":null,
      "subscriptionsPerConnection":5,
      "clientName":null,
      "loadBalancer":{
         "class":"org.redisson.connection.balancer.RoundRobinLoadBalancer"
      },
      "slaveSubscriptionConnectionMinimumIdleSize":1,
      "slaveSubscriptionConnectionPoolSize":25,
      "slaveConnectionMinimumIdleSize":5,
      "slaveConnectionPoolSize":100,
      "masterConnectionMinimumIdleSize":5,
      "masterConnectionPoolSize":100,
      "readMode":"SLAVE",
      "slaveAddresses":[
         "//127.0.0.1:6381",
         "//127.0.0.1:6380"
      ],
      "masterAddress":[
         "//127.0.0.1:6379"
      ],
```

```
        "database":0
    },
    "threads":0,
    "codec":null,
    "useLinuxNativeEpoll":false,
    "eventLoopGroup":null
}
```

以下是 YAML 格式的主从配置示例。

所有属性名称都匹配 `MasterSlaveServersConfig` 和 `Config` 对象的属性名称。

```
---
masterSlaveServersConfig:
  idleConnectionTimeout: 10000
  pingTimeout: 1000
  connectTimeout: 1000
  timeout: 1000
  retryAttempts: 3
  retryInterval: 1000
  reconnectionTimeout: 3000
  failedAttempts: 3
  password: null
  subscriptionsPerConnection: 5
  clientName: null
  loadBalancer: !
<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
  slaveSubscriptionConnectionMinimumIdleSize: 1
  slaveSubscriptionConnectionPoolSize: 25
  slaveConnectionMinimumIdleSize: 5
  slaveConnectionPoolSize: 100
  masterConnectionMinimumIdleSize: 5
  masterConnectionPoolSize: 100
  readMode: "SLAVE"
  slaveAddresses:
  - "//127.0.0.1:6381"
  - "//127.0.0.1:6380"
```

```
  masterAddress:
  - "//127.0.0.1:6379"
  database: 0
threads: 0
codec: !<org.redisson.codec.JsonJacksonCodec> {}
useLinuxNativeEpoll: false
eventLoopGroup: null
```

操作执行

操作执行

Redisson 支持对每个操作自动重试的策略并且在每次尝试期会尝试发送命令。
重试策略由设置项 `retryAttempts` （默认为 `3` ）和 `retryInterval` （默认为 `1000` ms）来控制。
每次尝试会在 `retryInterval` 时间间隔后执行。

Redisson 实例和 Redisson 对象都是完全线程安全的。

带有同步/异步方法的 Redisson 对象可通过 `RedissonClient` 接口获得。
替代的带有 Reactive Streams 方法的 Redisson 对象可通过 `RedissonReactiveClient` 接口获得。

以下是 `RAtomicLong` 对象的示例：

```
RedissonClient client = Redisson.create(config);
RAtomicLong longObject = client.getAtomicLong('myLong');
// sync way
longObject.compareAndSet(3, 401);
// async way
longObject.compareAndSetAsync(3, 401);

RedissonReactiveClient client = Redisson.createReactive(config);
RAtomicLongReactive longObject = client.getAtomicLong('myLong');
// reactive way
```

```
longObject.compareAndSet(3, 401);
```

异步方式

几乎每个 Redisson 对象都扩展了具有和同步方法的镜像的异步方法的一个异步接口。如：

```
// RAtomicLong extends RAtomicLongAsync
RAtomicLongAsync longObject = client.getAtomicLong("myLong");
Future<Boolean> future = longObject.compareAndSetAsync(1, 401);
```

异步方法返回一个扩展的具有可添加监听器的 `Future` 对象。
这样你可以以完全非阻塞的方式来获取结果。

```
future.addListener(new FutureListener<Boolean>() {
    @Override
    public void operationComplete(Future<Boolean> future) throws Exception {
          if (future.isSuccess()) {
             // get result
             Boolean result = future.getNow();
             // ...
          } else {
             // an error has occurred
             Throwable cause = future.cause();
          }
    }
});
```

Reactive 方式

Redisson 通过对 Java 9 的 Reactive Streams 标准来支持 Reactive 方式。
基于著名的 Reactor 项目。

针对 Java 的 Reactive 对象可通过独立的 `RedissonReactiveClient` 接口来获取：

```
1. RedissonReactiveClient client = Redisson.createReactive(config);
2. RAtomicLongReactive longObject = client.getAtomicLong("myLong");
3. 
4. Publisher<Boolean> csPublisher = longObject.compareAndSet(10, 91);
5. 
6. Publisher<Long> getPublisher = longObject.get();
```

数据序列化

数据序列化

数据序列化在 Redisson 中广泛地用于解编排在 Redis 服务器连接的网络上接收和发送的字节。
默认有多种流行的解编码器可用：

Codec 类名	描述
`org.redisson.codec.JsonJacksonCodec`	Jackson JSON codec. 默认 **codec**
`org.redisson.codec.CborJacksonCodec`	CBOR 二进制 json codec
`org.redisson.codec.MsgPackJacksonCodec`	MsgPack 二进制 json codec
`org.redisson.codec.KryoCodec`	Kryo 二进制 codec
`org.redisson.codec.SerializationCodec`	JDK 序列化 codec
`org.redisson.codec.FstCodec`	FST 10 倍速且 100% JDK 序列化兼容的 codec
`org.redisson.codec.LZ4Codec`	LZ4 压缩 codec
`org.redisson.codec.SnappyCodec`	Snappy 压缩 codec
`org.redisson.client.codec.StringCodec`	简单 String codec
`org.redisson.client.codec.LongCodec`	简单 Long codec

数据分区

数据分区(分片)

Redisson 仅在集群模式中支持数据分区(分片)。

它使得可以使用整个 Redis 集群的内存而不是单个节点的内存来存储单个数据结构实例。

Redisson 默认将数据结构切分为 **231** 个槽。
槽的数量可在 `3` 和 `16834` 之间。
槽会一致地分布在所有的集群节点上。
这意味着每个节点将包含近似相等数量的槽。
如默认槽量(231) 和 4 个节点的情况，每个节点将包含接近 57 个数据分区，
而对 5 个节点集群每个节点有 46 个数据分区等等。
这个特性基于 Redisson 中所使用的特殊的槽分布算法。

当前数据分区仅支持
`Set` 和
`Map`。

这个特性仅在 *Redisson Pro* 版本中存在。

分布式对象

分布式对象

每个 Redisson 对象都绑定到一个 Redis 键(即对象名称)，且可以通过 `getName` 方法读取。

```
1. RMap map = redisson.getMap("mymap");
2. map.getName(); // = mymap
```

所有和 Redis 键相关的操作被抽象到了 `RKeys` 接口：

```
1. RKeys keys = redisson.getKeys();
2.
3. Iterable<String> allKeys = keys.getKeys();
4. Iterable<String> foundedKeys = keys.getKeysByPattern('key*');
5. long numOfDeletedKeys = keys.delete("obj1", "obj2", "obj3");
6. long deletedKeysAmount = keys.deleteByPattern("test?");
7. String randomKey = keys.randomKey();
8. long keysAmount = keys.count();
```

TOC

Object

Redisson 分布式的 `RBucket` 对象可用作任意类型对象的通用容器。

```
RBucket<AnyObject> bucket = redisson.getBucket("anyObject");
bucket.set(new AnyObject(1));
AnyObject obj = bucket.get();

bucket.trySet(new AnyObject(3));
bucket.compareAndSet(new AnyObject(4), new AnyObject(5));
bucket.getAndSet(new AnyObject(6));
```

地理位置容器

Redisson 分布式的 `RGeo` 对象 可用作地理位置项的容器。

```
RGeo<String> geo = redisson.getGeo("test");
geo.add(new GeoEntry(13.361389, 38.115556, "Palermo"),
        new GeoEntry(15.087269, 37.502669, "Catania"));
geo.addAsync(37.618423, 55.751244, "Moscow");

Double distance = geo.dist("Palermo", "Catania", GeoUnit.METERS);
geo.hashAsync("Palermo", "Catania");
Map<String, GeoPosition> positions = geo.pos("test2", "Palermo",
"test3", "Catania", "test1");
List<String> cities = geo.radius(15, 37, 200, GeoUnit.KILOMETERS);
```

```
10. Map<String, GeoPosition> citiesWithPositions =
    geo.radiusWithPosition(15, 37, 200, GeoUnit.KILOMETERS);
```

BitSet

Redisson 分布式的 `RBitSet` 对象具有类似于 `java.util.BitSet` 的结构，
且表示的位向量会根据需要增长。BitSet 的大小由 Redis 限制为 `4 294 967 295` 。

```
1. RBitSet set = redisson.getBitSet("simpleBitset");
2. set.set(0, true);
3. set.set(1812, false);
4. set.clear(0);
5. set.addAsync("e");
6. set.xor("anotherBitset");
```

AtomicLong

Redisson 分布式的 `RAtomicLong` 对象具有类似于 `java.util.concurrent.atomic.AtomicLong` 对象的结构。

```
1. RAtomicLong atomicLong = redisson.getAtomicLong("myAtomicLong");
2. atomicLong.set(3);
3. atomicLong.incrementAndGet();
4. atomicLong.get();
```

AtomicDouble

Redisson 分布式的 `AtomicDouble` 对象.

```
1. RAtomicDouble atomicDouble =
    redisson.getAtomicDouble("myAtomicDouble");
```

```
atomicDouble.set(2.81);
atomicDouble.addAndGet(4.11);
atomicDouble.get();
```

Topic

Redisson 分布式的 `Topic` 对象实现了 发布/订阅 机制。

```
RTopic<SomeObject> topic = redisson.getTopic("anyTopic");
topic.addListener(new MessageListener<SomeObject>() {
    @Override
    public void onMessage(String channel, SomeObject message) {
        //...
    }
});

// in other thread or JVM
RTopic<SomeObject> topic = redisson.getTopic("anyTopic");
long clientsReceivedMessage = topic.publish(new SomeObject());
```

Topic 模式

Redisson Topic pattern 对象可通过指定模式订阅到多个 topics。

```
// subscribe to all topics by `topic1.*` pattern
RPatternTopic<Message> topic1 =
redisson.getPatternTopic("topic1.*");
int listenerId = topic1.addListener(new
PatternMessageListener<Message>() {
    @Override
    public void onMessage(String pattern, String channel, Message
msg) {
         Assert.fail();
    }
```

```
8. });
```

Topic 模式监听器在重连到 Redis 服务器或者 Redis 服务器故障恢复时自动重新订阅。

Bloom filter

Redisson 分布式的 Bloom filter 对象。

```
1. RBloomFilter<SomeObject> bloomFilter =
   redisson.getBloomFilter("sample");
2. // initialize bloom filter with
3. // expectedInsertions = 55000000
4. // falseProbability = 0.03
5. bloomFilter.tryInit(55000000L, 0.03);
6. bloomFilter.add(new SomeObject("field1Value", "field2Value"));
7. bloomFilter.add(new SomeObject("field5Value", "field8Value"));
8. bloomFilter.contains(new SomeObject("field1Value", "field8Value"));
```

HyperLogLog

Redisson 分布式的 HyperLogLog 对象。

```
1. RHyperLogLog<Integer> log = redisson.getHyperLogLog("log");
2. log.add(1);
3. log.add(2);
4. log.add(3);
5.
6. log.count();
```

远程服务

基于 Redis 的分布式 Java 远程服务使得可以在不同的 Redisson 实例上执行远程接口上的对象方法。

或者说它可以执行 Java 远程调用。
使用 POJO 对象，方法参数和结果对象类型可以是任意的。

`RemoteService` 提供了两种类型的 `RRemoteService` 实例：

- 服务端实例 - 执行远程方法(工作者实例)，如：

```
RRemoteService remoteService = redisson.getRemoteService();
SomeServiceImpl someServiceImpl = new SomeServiceImpl();

// register remote service before any remote invocation
// can handle only 1 invocation concurrently
remoteService.register(SomeServiceInterface.class,
someServiceImpl);

// register remote service able to handle up to 12 invocations
concurrently
remoteService.register(SomeServiceInterface.class, someServiceImpl,
12);
```

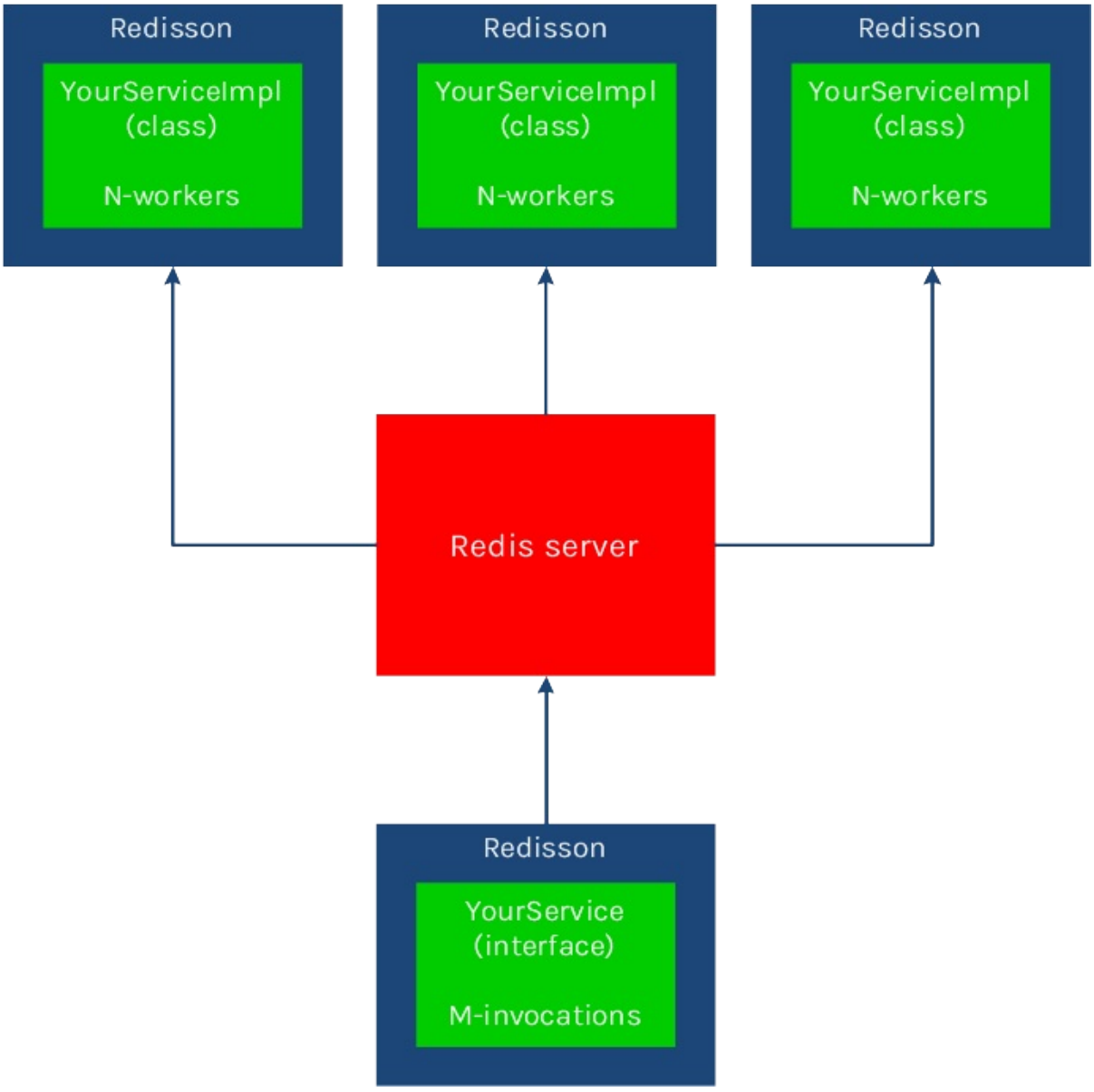
- 客户端实例 - 调用远程方法。如：

```
RRemoteService remoteService = redisson.getRemoteService();
SomeServiceInterface service =
remoteService.get(SomeServiceInterface.class);

String result = service.doSomeStuff(1L, "secondParam", new
AnyParam());
```

客户端和服务端实例应使用相同的远程接口且后端是使用相同的服务器连接配置所创建的 Redisson 实例。
客户端和服务端实例可运行在相同的 JVM 上。
对客户端和/或服务端实例的数量没有限制。
(注：尽管 Redisson 没有强制任何限制，但 Redis 本事的限制还是存在的。)

远程调用在 `1+` 个工作者存在时会以 并行模式 执行。



并行执行器总量可计算如下：

`T` = `R` * `N`

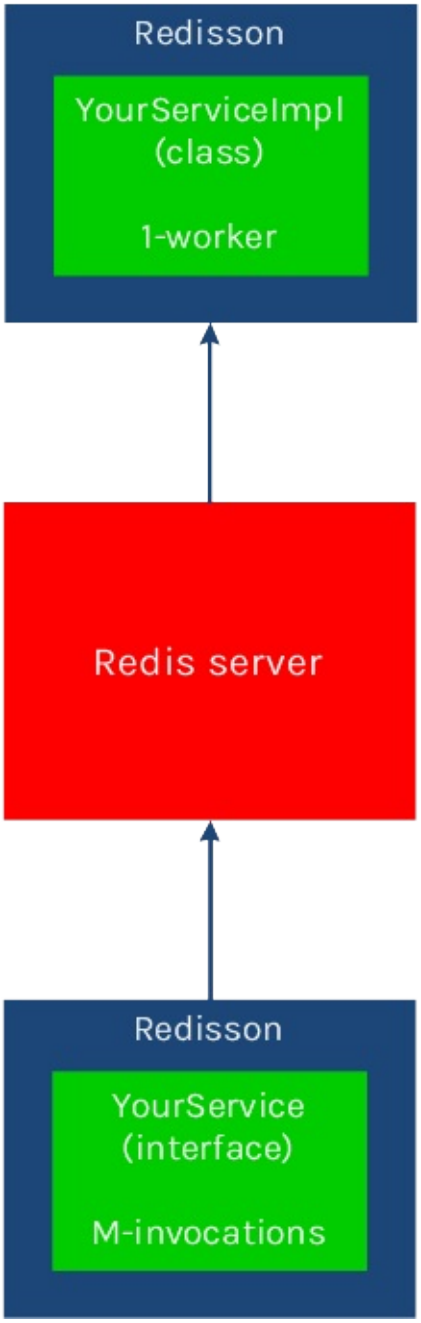
`T` - 总可用并行执行器数量

`R` - Redisson 服务端实例数量

`N` - 在服务注册时所定义的执行器数量

超过这个数量的命令将放到队列中，在后续有可用执行器时再执行。

远程调用在 仅有 `1` 个工作者存在时将以 顺序模式 执行。
这时仅有一个命令会被同时执行，其它的命令都会被放到队列中。

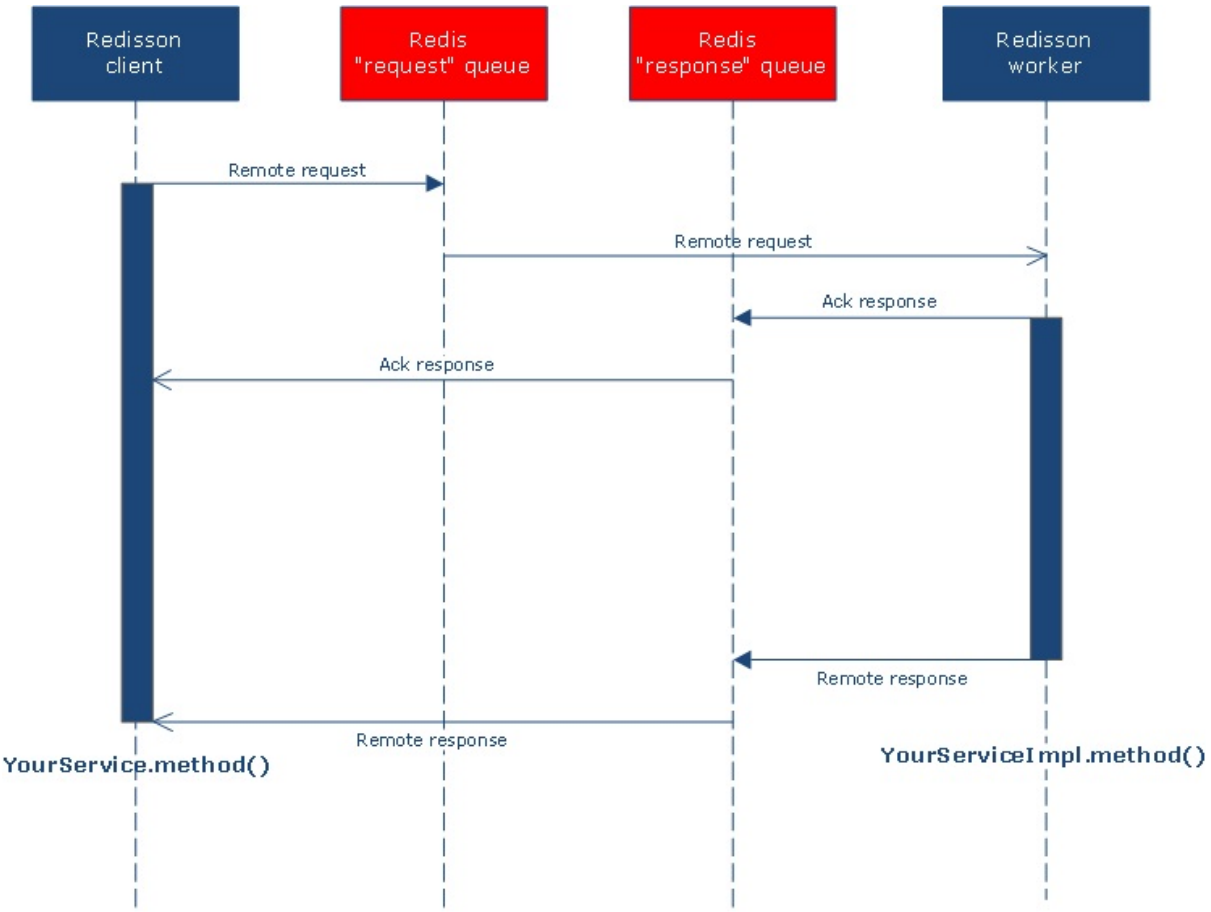


远程服务消息流

`RemoteService` 对每个调用创建了两个队列。
一个队列用于请求(由服务端实例监听)，而另一个用于应答响应和结构响应(由客户端实例监听)。

应答响应用于确定方法执行器是否有一个请求。
若在超时时间内没有得到应答，将抛出一个
`RemoteServiceAckTimeoutException` 。

下图描绘了每个远程调用的消息流：



Remote service. Fire-and-forget and ack-response modes

RemoteService 可通过 `org.redisson.core.RemoteInvocationOptions` 对象来
为每次远程调用提供选项。
这些选项可用于改变超时时间和跳过应答响应和/或结果响应。如：

```
1.  // 1 second ack timeout and 30 seconds execution timeout
```

```
RemoteInvocationOptions options =
RemoteInvocationOptions.defaults();

// no ack but 30 seconds execution timeout
RemoteInvocationOptions options =
RemoteInvocationOptions.defaults().noAck();

// 1 second ack timeout then forget the result
RemoteInvocationOptions options =
RemoteInvocationOptions.defaults().noResult();

// 1 minute ack timeout then forget about the result
RemoteInvocationOptions options =
RemoteInvocationOptions.defaults().expectAckWithin(1,
TimeUnit.MINUTES).noResult();

// no ack and forget about the result (fire and forget)
RemoteInvocationOptions options =
RemoteInvocationOptions.defaults().noAck().noResult();

RRemoteService remoteService = redisson.getRemoteService();
YourService service = remoteService.get(YourService.class,
options);
```

远程服务异步调用

远程调用可以异步方式执行。
这时应该使用单独的以 `@RRemoteAsync` 注解标注的接口。
其方法签名需匹配远程接口中的相同的方法。
每个方法应返回 `io.netty.util.concurrent.Future` 对象。
异步接口验证将在 `RRemoteService.get` 方法调用期间被执行。
并不需要把所有方法都列出来，只需要列出需要以异步方式调用的方法。

```
public interface RemoteInterface {
```

```
2.
3.      Long someMethod1(Long param1, String param2);
4.
5.      void someMethod2(MyObject param);
6.
7.      MyObject someMethod3();
8.
9.  }
10.
11. // async interface for RemoteInterface
12. @RRemoteAsync(RemoteInterface.class)
13. public interface RemoteInterfaceAsync {
14.
15.     Future<Long> someMethod1(Long param1, String param2);
16.
17.     Future<Void> someMethod2(MyObject param);
18.
19. }
20.
21. RRemoteService remoteService = redisson.getRemoteService();
22. RemoteInterfaceAsync asyncService =
    remoteService.get(RemoteInterfaceAsync.class);
```

分布式集合

分布式集合

TOC

Map

Redisson 分布式的 Map 对象，实现了 `java.util.concurrent.ConcurrentMap` 和 `java.util.Map` 接口。

Map 的大小由 Redis 限制为 `4 294 967 295`。

```
RMap<String, SomeObject> map = redisson.getMap("anyMap");
SomeObject prevObject = map.put("123", new SomeObject());
SomeObject currentObject = map.putIfAbsent("323", new
SomeObject());
SomeObject obj = map.remove("123");

map.fastPut("321", new SomeObject());
map.fastRemove("321");

Future<SomeObject> putAsyncFuture = map.putAsync("321");
Future<Void> fastPutAsyncFuture = map.fastPutAsync("321");

map.fastPutAsync("321", new SomeObject());
map.fastRemoveAsync("321");
```

Redisson PRO 版本的 Map 对象
在集群模式中支持 数据分区。

Map eviction

Redisson 分布式的 Map 可通过独立的 MapCache 对象支持 eviction。
它也实现了 `java.util.concurrent.ConcurrentMap`
和 `java.util.Map` 接口。
Redisson 有一个基于 Map 和 MapCache 对象的
Spring Cache 集成。

当前 Redis 实现中没有 map 项的 eviction 功能。
因此，过期的项由 `org.redisson.EvictionScheduler` 来清理。
它一次可移除 100 条过期项。
任务的调度时间会根据上次任务中删除的过期项数量自动调整，时间在

1 秒到 2 个小时内。
因此若清理任务每次删除了 100 项数据，它将每秒钟执行一次(最小的执行延迟)。
但如果当前过期项数量比前一次少，则执行延迟将扩大为 1.5 倍。

```
RMapCache<String, SomeObject> map = redisson.getMapCache("anyMap");
// ttl = 10 minutes,
map.put("key1", new SomeObject(), 10, TimeUnit.MINUTES);
// ttl = 10 minutes, maxIdleTime = 10 seconds
map.put("key1", new SomeObject(), 10, TimeUnit.MINUTES, 10,
TimeUnit.SECONDS);

// ttl = 3 seconds
map.putIfAbsent("key2", new SomeObject(), 3, TimeUnit.SECONDS);
// ttl = 40 seconds, maxIdleTime = 10 seconds
map.putIfAbsent("key2", new SomeObject(), 40, TimeUnit.SECONDS, 10,
TimeUnit.SECONDS);
```

MultiMap

Redisson 分布式的 MultiMap 对象允许给每个键绑定多个值。
键的数量限制由 Redis 限制为 `4 294 967 295` 。

基于 Set 的 MultiMap

基于 Set 的 MultiMap 不允许每个键中的值有重复。

```
RSetMultimap<SimpleKey, SimpleValue> map =
redisson.getSetMultimap("myMultimap");
map.put(new SimpleKey("0"), new SimpleValue("1"));
map.put(new SimpleKey("0"), new SimpleValue("2"));
map.put(new SimpleKey("3"), new SimpleValue("4"));

Set<SimpleValue> allValues = map.get(new SimpleKey("0"));

```

```
List<SimpleValue> newValues = Arrays.asList(new SimpleValue("7"),
new SimpleValue("6"), new SimpleValue("5"));
Set<SimpleValue> oldValues = map.replaceValues(new SimpleKey("0"),
newValues);

Set<SimpleValue> removedValues = map.removeAll(new SimpleKey("0"));
```

基于 List 的 MultiMap

基于 List 的 MultiMap 会存储插入的顺序且允许键对应的值中存在重复。

```
RListMultimap<SimpleKey, SimpleValue> map =
redisson.getListMultimap("test1");
map.put(new SimpleKey("0"), new SimpleValue("1"));
map.put(new SimpleKey("0"), new SimpleValue("2"));
map.put(new SimpleKey("0"), new SimpleValue("1"));
map.put(new SimpleKey("3"), new SimpleValue("4"));

List<SimpleValue> allValues = map.get(new SimpleKey("0"));

Collection<SimpleValue> newValues = Arrays.asList(new
SimpleValue("7"), new SimpleValue("6"), new SimpleValue("5"));
List<SimpleValue> oldValues = map.replaceValues(new SimpleKey("0"),
newValues);

List<SimpleValue> removedValues = map.removeAll(new
SimpleKey("0"));
```

MultiMap eviction

Multimap 对象可通过独立的 MultimapCache 对象来支持 eviction。
它对基于 Set 和 List 的 MultiMap 分别有 `RSetMultimapCache` 和 `RListMultimapCache` 对象。

过期的项由 `org.redisson.EvictionScheduler` 来清理。
它一次可移除 100 条过期项。
任务的调度时间会根据上次任务中删除的过期项数量自动调整，时间在 1 秒到 2 个小时内。
因此若清理任务每次删除了 100 项数据，它将每秒钟执行一次(最小的执行延迟)。
但如果当前过期项数量比前一次少，则执行延迟将扩大为 1.5 倍。

RSetMultimapCache 示例:

```
1. RSetMultimapCache<String, String> multimap =
   redisson.getSetMultimapCache("myMultimap");
2. map.put("1", "a");
3. map.put("1", "b");
4. map.put("1", "c");
5.
6. map.put("2", "e");
7. map.put("2", "f");
8.
9. map.expireKey("2", 10, TimeUnit.MINUTES);
```

Set

Redisson 分布式的 Set 对象，实现了 `java.util.Set` 接口。
它通过元素状态比较来保持元素的独立性。
Set 大小由 Redis 限制为 `4 294 967 295` 。

```
1. RSet<SomeObject> set = redisson.getSet("anySet");
2. set.add(new SomeObject());
3. set.remove(new SomeObject());
```

Redisson PRO 版本的 Set 对象
在集群模式中支持 数据分区。

Set eviction

Redisson 分布式的 Set 对象可通过独立的 SetCache 对象来支持 eviction。
它也实现了 `java.util.Set` 接口。

当前 Redis 实现中没有 set 值的 eviction 功能。
因此，过期的项由 `org.redisson.EvictionScheduler` 来清理。
它一次可移除 100 条过期项。
任务的调度时间会根据上次任务中删除的过期项数量自动调整，时间在 1 秒到 2 个小时内。
因此若清理任务每次删除了 100 项数据，它将每秒钟执行一次(最小的执行延迟)。
但如果当前过期项数量比前一次少，则执行延迟将扩大为 1.5 倍。

```
1.  RSetCache<SomeObject> set = redisson.getSetCache("anySet");
2.  // ttl = 10 seconds
3.  set.add(new SomeObject(), 10, TimeUnit.SECONDS);
```

SortedSet

Redisson 分布式的 SortedSet 对象，实现了 `java.util.SortedSet` 接口。
它通过比较器来排列元素并保持唯一性。

```
1.  RSortedSet<Integer> set = redisson.getSortedSet("anySet");
2.  set.trySetComparator(new MyComparator()); // set object comparator
3.  set.add(3);
4.  set.add(1);
5.  set.add(2);
6.
7.  set.removeAsync(0);
```

```
8.  set.addAsync(5);
```

ScoredSortedSet

Redisson 分布式的 ScoredSortedSet 对象。
它通过在元素插入时定义的分数来排列元素，
通过元素状态的比较来保持元素的唯一性。

```
1.  RScoredSortedSet<SomeObject> set =
    redisson.getScoredSortedSet("simple");
2.
3.  set.add(0.13, new SomeObject(a, b));
4.  set.addAsync(0.251, new SomeObject(c, d));
5.  set.add(0.302, new SomeObject(g, d));
6.
7.  set.pollFirst();
8.  set.pollLast();
9.
10. int index = set.rank(new SomeObject(g, d)); // get element index
11. Double score = set.getScore(new SomeObject(g, d)); // get element
    score
```

LexSortedSet

Redisson 分布式的 Set 对象，它仅允许字典序的 String 对象，
并实现了 `java.util.Set<String>` 接口。
它通过元素状态比较来保持元素的唯一性。

```
1.  RLexSortedSet set = redisson.getLexSortedSet("simple");
2.  set.add("d");
3.  set.addAsync("e");
4.  set.add("f");
5.
6.  set.lexRangeTail("d", false);
```

```
7.  set.lexCountHead("e");
8.  set.lexRange("d", true, "z", false);
```

List

Redisson 分布式的 List 对象，实现了 `java.util.List` 接口。
它保留元素的插入顺序。
List 大小由 Redis 限制为 `4 294 967 295` 。

```
1.  RList<SomeObject> list = redisson.getList("anyList");
2.  list.add(new SomeObject());
3.  list.get(0);
4.  list.remove(new SomeObject());
```

Queue

Redisson 分布式的 Queue 对象，实现了 `java.util.Queue` 接口。
Queue 大小由 Redis 限制为 `4 294 967 295` 。

```
1.  RQueue<SomeObject> queue = redisson.getQueue("anyQueue");
2.  queue.add(new SomeObject());
3.  SomeObject obj = queue.peek();
4.  SomeObject someObj = queue.poll();
```

Deque

Redisson 分布式的 Deque 对象，实现了 `java.util.Deque` 接口。
Deque 大小由 Redis 限制为 `4 294 967 295` 。

```
1.  RDeque<SomeObject> queue = redisson.getDeque("anyDeque");
```

```
queue.addFirst(new SomeObject());
queue.addLast(new SomeObject());
SomeObject obj = queue.removeFirst();
SomeObject someObj = queue.removeLast();
```

BlockingQueue

Redisson 分布式的 BlockingQueue 对象，实现了 `java.util.concurrent.BlockingQueue` 接口。
BlockingQueue 大小由 Redis 限制为 `4 294 967 295`。

```
RBlockingQueue<SomeObject> queue =
redisson.getBlockingQueue("anyQueue");
queue.offer(new SomeObject());

SomeObject obj = queue.peek();
SomeObject someObj = queue.poll();
SomeObject ob = queue.poll(10, TimeUnit.MINUTES);
```

`poll`、`pollFromAny`、`pollLastAndOfferFirstTo` 和 `take` 方法在重连到 Redis 服务器或 Redis 服务器故障恢复时会自动被重新订阅。

BlockingDeque

Redisson 分布式的 BlockingDeque 对象，实现了 `java.util.concurrent.BlockingDeque` 接口。
BlockingDeque 大小由 Redis 限制为 `4 294 967 295`。

```
RBlockingDeque<Integer> deque =
redisson.getBlockingDeque("anyDeque");
deque.putFirst(1);
deque.putLast(2);
```

```
Integer firstValue = queue.takeFirst();
Integer lastValue = queue.takeLast();
Integer firstValue = queue.pollFirst(10, TimeUnit.MINUTES);
Integer lastValue = queue.pollLast(3, TimeUnit.MINUTES);
```

`poll`、`pollFromAny`、`pollLastAndOfferFirstTo` 和 `take` 方法在重连到 Redis 服务器或 Redis 服务器故障恢复时会自动被重新订阅。

分布式锁和同步器

分布式锁和同步器

TOC

Lock

Redisson 分布式可重入锁，实现了 `java.util.concurrent.locks.Lock` 接口并支持 TTL。

```
1.  RLock lock = redisson.getLock("anyLock");
2.  // Most familiar locking method
3.  lock.lock();
4.
5.  // Lock time-to-live support
6.  // releases lock automatically after 10 seconds
7.  // if unlock method not invoked
8.  lock.lock(10, TimeUnit.SECONDS);
9.
10. // Wait for 100 seconds and automatically unlock it after 10
    seconds
11. boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
12. ...
13. lock.unlock();
```

Redisson 也支持 Lock 对象的异步方法：

```
RLock lock = redisson.getLock("anyLock");
lock.lockAsync();
lock.lockAsync(10, TimeUnit.SECONDS);
Future<Boolean> res = lock.tryLockAsync(100, 10, TimeUnit.SECONDS);
```

Fair Lock

Redisson 分布式可重入公平锁，实现了 `java.util.concurrent.locks.Lock` 接口并支持 TTL，并且保证 Redisson 客户端线程将以其请求的顺序获得锁。它和简单的 Lock 对象有相同的接口。

```
RLock fairLock = redisson.getFairLock("anyLock");
// Most familiar locking method
fairLock.lock();

// Lock time-to-live support
// releases lock automatically after 10 seconds
// if unlock method not invoked
fairLock.lock(10, TimeUnit.SECONDS);

// Wait for 100 seconds and automatically unlock it after 10 seconds
boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);
...
fairLock.unlock();
```

Redisson 也支持公平锁对象的异步方法：

```
RLock fairLock = redisson.getFairLock("anyLock");
fairLock.lockAsync();
fairLock.lockAsync(10, TimeUnit.SECONDS);
```

```
4.  Future<Boolean> res = fairLock.tryLockAsync(100, 10,
    TimeUnit.SECONDS);
```

MultiLock

`RedissonMultiLock` 对象可用于实现 Redlock 锁算法。
它将多个 `RLock` 对象划为一组并且将它们当作一个锁来处理。
每个 `RLock` 对象可以属于不同的 Redisson 实例。

```
1.  RLock lock1 = redissonInstance1.getLock("lock1");
2.  RLock lock2 = redissonInstance2.getLock("lock2");
3.  RLock lock3 = redissonInstance3.getLock("lock3");
4.
5.  RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2,
    lock3);
6.  // locks: lock1 lock2 lock3
7.  lock.lock();
8.  ...
9.  lock.unlock();
```

ReadWriteLock

Redisson 分布式可重入 ReadWriteLock 对象，实现了 `java.util.concurrent.locks.ReadWriteLock` 接口并支持 TTL。
可以同时存在多个 ReadLock 拥有者，但仅允许有一个 `WriteLock`

```
1.  RReadWriteLock rwlock = redisson.getLock("anyRWLock");
2.  // Most familiar locking method
3.  rwlock.readLock().lock();
4.  // or
5.  rwlock.writeLock().lock();
6.
7.  // Lock time-to-live support
8.  // releases lock automatically after 10 seconds
```

```
// if unlock method not invoked
rwlock.readLock().lock(10, TimeUnit.SECONDS);
// or
rwlock.writeLock().lock(10, TimeUnit.SECONDS);

// Wait for 100 seconds and automatically unlock it after 10 seconds
boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);
// or
boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);
...
lock.unlock();
```

Semaphore

Redisson 分布式 Semaphore 对象，类似于 `java.util.concurrent.Semaphore` 对象。

```
RSemaphore semaphore = redisson.getSemaphore("semaphore");
semaphore.acquire();
//or
semaphore.acquireAsync();
semaphore.acquire(23);
semaphore.tryAcquire();
//or
semaphore.tryAcquireAsync();
semaphore.tryAcquire(23, TimeUnit.SECONDS);
//or
semaphore.tryAcquireAsync(23, TimeUnit.SECONDS);
semaphore.release(10);
semaphore.release();
//or
semaphore.releaseAsync();
```

CountDownLatch

Redisson 分布式 CountDownLatch 对象，结构类似于 `java.util.concurrent.CountDownLatch` 对象。

```
RCountDownLatch latch =
redisson.getCountDownLatch("anyCountDownLatch");
latch.trySetCount(1);
latch.await();

// in other thread or other JVM
RCountDownLatch latch =
redisson.getCountDownLatch("anyCountDownLatch");
latch.countDown();
```

其它特性

其它特性

操作节点

Redisson NodesGroup 对象提供了对 Redis 节点的一些控制：

```
NodesGroup nodesGroup = redisson.getNodesGroup();
nodesGroup.addConnectionListener(new ConnectionListener() {
    public void onConnect(InetSocketAddress addr) {
       // Redis server connected
    }

    public void onDisconnect(InetSocketAddress addr) {
       // Redis server disconnected
    }
});
```

允许给单个或所有 Redis 服务器发送 ping 请求：

```
NodesGroup nodesGroup = redisson.getNodesGroup();
Collection<Node> allNodes = nodesGroup.getNodes();
for (Node n : allNodes) {
    n.ping();
}
```

```
// or
nodesGroup.pingAll();
```

执行批量命令

通过 `RBatch` 对象可以将多个命令汇总到一个网络调用中一次性发送并执行。
通过这个对象你可以一组命令的执行时间。
在 Redis 中这种方式称为 Pipeling。

```
RBatch batch = redisson.createBatch();
batch.getMap("test").fastPutAsync("1", "2");
batch.getMap("test").fastPutAsync("2", "3");
batch.getMap("test").putAsync("2", "5");
batch.getAtomicLongAsync("counter").incrementAndGetAsync();
batch.getAtomicLongAsync("counter").incrementAndGetAsync();

List<?> res = batch.execute();
```

脚本

```
redisson.getBucket("foo").set("bar");
String r = redisson.getScript().eval(Mode.READ_ONLY,
   "return redis.call('get', 'foo')", RScript.ReturnType.VALUE);

// do the same using cache
RScript s = redisson.getScript();
// load script into cache to all redis master instances
String res = s.scriptLoad("return redis.call('get', 'foo')");
// res == 282297a0228f48cd3fc6a55de6316f31422f5d17

// call script by sha digest
Future<Object> r1 =
redisson.getScript().evalShaAsync(Mode.READ_ONLY,
   "282297a0228f48cd3fc6a55de6316f31422f5d17",
```

```
14.     RScript.ReturnType.VALUE, Collections.emptyList());
```

Spring Cache 集成

Redisson 完全支持 Spring Cache 抽象.
每个 Cache 实例具有两个重要参数： `ttl` 和 `maxIdleTime`，
且在它们没有定义或者等于 `0` 时会永久存储。
配置示例：

```
1.  @Configuration
2.  @ComponentScan
3.  @EnableCaching
4.  public static class Application {
5.
6.      @Bean(destroyMethod="shutdown")
7.      RedissonClient redisson() throws IOException {
8.          Config config = new Config();
9.          config.useClusterServers()
10.                .addNodeAddress("127.0.0.1:7004", "127.0.0.1:7001");
11.         return Redisson.create(config);
12.     }
13.
14.     @Bean
15.     CacheManager cacheManager(RedissonClient redissonClient) {
16.         Map<String, CacheConfig> config = new HashMap<String,
    CacheConfig>();
17.         config.put("testMap", new CacheConfig(24*60*1000,
    12*60*1000));
18.         return new RedissonSpringCacheManager(redissonClient,
    config);
19.     }
20.
21. }
```

Cache 配置也可以从 JSON 或 YAML 配置文件中读取：

```
@Configuration
@ComponentScan
@EnableCaching
public static class Application {

    @Bean(destroyMethod="shutdown")
    RedissonClient redisson(@Value("classpath:/redisson.json") Resource configFile) throws IOException {
        Config config = Config.fromJSON(configFile.getInputStream());
        return Redisson.create(config);
    }

    @Bean
    CacheManager cacheManager(RedissonClient redissonClient) throws IOException {
        return new RedissonSpringCacheManager(redissonClient, "classpath:/cache-config.json");
    }

}
```

底层 Redis 客户端

Redisson 使用高性能异步且无锁的 Redis 客户端。
它支持异步和同步模式。
如果 Redisson 尚不支持，你可以通过它自行执行命令。
当然，在使用底层客户端之前，你可能想在 Redis命令映射 查询一下。

```
RedisClient client = new RedisClient("localhost", 6379);
RedisConnection conn = client.connect();
//or
Future<RedisConnection> connFuture = client.connectAsync();

```

```
conn.sync(StringCodec.INSTANCE, RedisCommands.SET, "test", 0);
conn.async(StringCodec.INSTANCE, RedisCommands.GET, "test");

conn.sync(RedisCommands.PING);

conn.close()
// or
conn.closeAsync()

client.shutdown();
// or
client.shutdownAsync();
```

Redis命令映射

Redis命令映射

详见:

Redis 命令映射