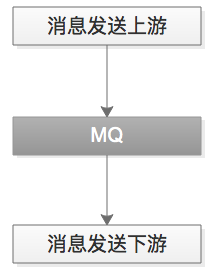
# Redis消息队列与数据库

## 1. 消息队列

### 1. 定义

消息队列（MQ）是一种应用程序对应用程序的通信方法。应用程序通过写和检索出入列队的针对应用程序的数据（消息）来通信，而无需专用连接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于诸如[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8)的技术。排队指的是应用程序通过队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。

消息总线（Message Queue），后文称MQ，是一种跨进程的通信机制，用于上下游传递消息。



在互联网架构中，MQ是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。

使用了MQ之后，消息发送上游只需要依赖MQ，逻辑上和物理上都不用依赖其他服务。

### 2. 分类

一般来说，消息队列有两种场景，一种是发布者订阅者模式，一种是生产者消费者模式。利用redis这两种场景的消息队列都能够实现。

**生产者消费者模式**：生产者生产消息放到队列里，多个消费者同时监听队列，谁先抢到消息谁就会从队列中取走消息；即对于每个消息只能被最多一个消费者拥有。

**发布者订阅者模式**：发布者生产消息放到队列里，多个监听队列的消费者都会收到同一份消息；即正常情况下每个消费者收到的消息应该都是一样的。

### 3.不适用场景

**调用方实时依赖执行结果的业务场景，请使用调用，而不是MQ。**

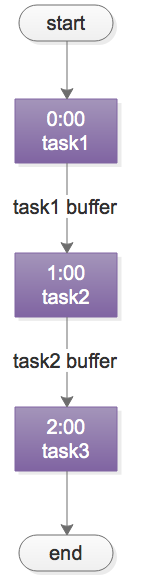
### 4. 适用场景

http://blog.csdn.net/xybelieve1990/article/details/70313216

#### 典型场景一：数据驱动的任务依赖

互联网公司经常在凌晨进行一些数据统计任务，这些任务之间有一定的依赖关系，比如：

1）task3需要使用task2的输出作为输入

2）task2需要使用task1的输出作为输入

这样的话，tast1, task2, task3之间就有任务依赖关系，必须task1先执行，再task2执行，载task3执行。

对于这类需求，常见的实现方式是，使用cron人工排执行时间表：

1）task1，0:00执行，经验执行时间为50分钟

2）task2，1:00执行（为task1预留10分钟buffer），经验执行时间也是50分钟

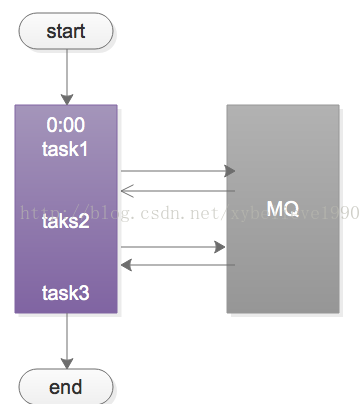
3）task3，2:00执行（为task2预留10分钟buffer）

这种方法的坏处是：

1）如果有一个任务执行时间超过了预留buffer的时间，将会得到错误的结果，因为后置任务不清楚前置任务是否执行成功，此时要手动重跑任务，还有可能要调整排班表

2）总任务的执行时间很长，总是要预留很多buffer，如果前置任务提前完成，后置任务不会提前开始

3）如果一个任务被多个任务依赖，这个任务将会称为关键路径，排班表很难体现依赖关系，容易出错

4）如果有一个任务的执行时间要调整，将会有多个任务的执行时间要调整

优化方案是，采用MQ解耦：

1）task1准时开始，结束后发一个“task1 done”的消息

2）task2订阅“task1 done”的消息，收到消息后第一时间启动执行，结束后发一个“task2 done”的消息

3）task3同理

采用MQ的优点是：

1）不需要预留buffer，上游任务执行完，下游任务总会在第一时间被执行

2）依赖多个任务，被多个任务依赖都很好处理，只需要订阅相关消息即可

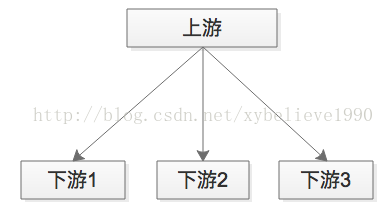
3）有任务执行时间变化，下游任务都不需要调整执行时间

需要特别说明的是，MQ只用来传递上游任务执行完成的消息，并不用于传递真正的输入输出数据。

#### 典型场景二：上游不关心执行结果

上游需要关注执行结果时要用“调用”，上游不关注执行结果时，就可以使用MQ了。

举个栗子，58同城的很多下游需要关注“用户发布帖子”这个事件，比如招聘用户发布帖子后，招聘业务要奖励58豆，房产用户发布帖子后，房产业务要送2个置顶，二手用户发布帖子后，二手业务要修改用户统计数据。



对于这类需求，常见的实现方式是，使用调用关系：

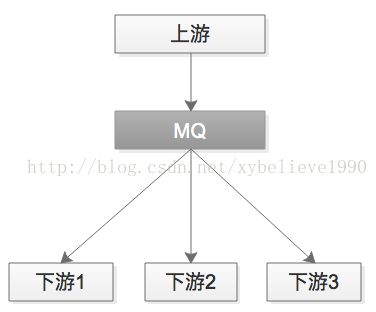
帖子发布服务执行完成之后，调用下游招聘业务、房产业务、二手业务，来完成消息的通知，但事实上，这个通知是否正常正确的执行，帖子发布服务根本不关注。

这种方法的坏处是：

1）帖子发布流程的执行时间增加了

2）下游服务当机，可能导致帖子发布服务受影响，上下游逻辑+物理依赖严重

3）每当增加一个需要知道“帖子发布成功”信息的下游，修改代码的是帖子发布服务，这一点是最恶心的，属于架构设计中典型的依赖倒转



优化方案是，采用MQ解耦：

1）帖子发布成功后，向MQ发一个消息

2）哪个下游关注“帖子发布成功”的消息，主动去MQ订阅

采用MQ的优点是：

1）上游执行时间短

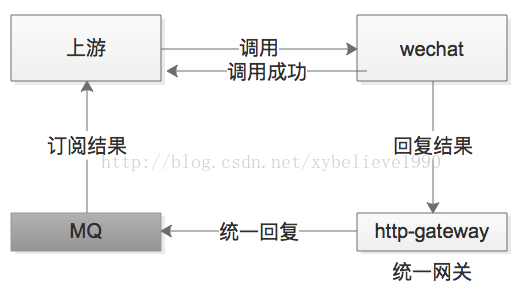
2）上下游逻辑+物理解耦，除了与MQ有物理连接，模块之间都不相互依赖

3）新增一个下游消息关注方，上游不需要修改任何代码

#### 典型场景三：上游关注执行结果，但执行时间很长

 有时候上游需要关注执行结果，但执行结果时间很长（典型的是调用离线处理，或者跨公网调用），也经常使用回调网关+MQ来解耦。

举个栗子，微信支付，跨公网调用微信的接口，执行时间会比较长，但调用方又非常关注执行结果，此时一般怎么玩呢？



一般采用“回调网关+MQ”方案来解耦：

1）调用方直接跨公网调用微信接口

2）微信返回调用成功，此时并不代表返回成功

3）微信执行完成后，回调统一网关

4）网关将返回结果通知MQ

5）请求方收到结果通知

这里需要注意的是，不应该由回调网关来调用上游来通知结果，如果是这样的话，每次新增调用方，回调网关都需要修改代码，仍然会反向依赖，使用回调网关+MQ的方案，新增任何对微信支付的调用，都不需要修改代码。

### 5. **Redis是内存数据库**

#### ****内存数据库****

Redis，本质上上一个KEY-VALUE类型的内存数据库，整个数据库都加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因此它是纯内存操作，Redis的性能非常出色，每秒可以处理超过10万次读写操作。虽然是内存数据库，但是其数据可以持久化，而且支持丰富的数据类型。

Redis支持保存LIST列表和SET集合的数据结构，而且还支持对LIST进行各种操作，例如从LIST两端进行PUSH和POP数据，取LIST区间，排序等等。对SET支持各种集合的并集交集操作，单个value的最大限制是1GB。

Redis主要的缺点是受到物理内存限制，不能用作海量数据的高性能读写，而且它没有原生的可扩展机制，不具有扩展能力，要依赖客户端来实现分布式读写，因此其适合的应用场景主要局限在较小数据量的高性能操作和运算上。

**最佳应用场景：适用于数据变化快且数据库大小可遇见（内存大小）的应用程序。**

*例如：股票软件、数据分析、实时数据收集、实时通讯。*

Redis属于NoSQL范畴内，其含义为：Not only SQL，是不仅仅是SQL，是一项新的技术，随着WEB 2. 0的到来而得到广泛应用。理念是运用非关系的数据存储。

键值类型的数据库主要使用哈希表，这个表中有一个特定的键和一个指针指向特定数据。KEY/VALUE模型对于IT系统来说的优势在于简单、容易部署。主要特点是具有极高的并发读写性能。

#### 持久化

Redis的所有数据都保存在内存中，然后不定期的通过异步方式保存到磁盘上（这称为半持久化）；也可以把每一次数据变化都写入到磁盘（这称为全持久化）。所谓持久化就是将内存数据转换为硬盘数据，内存模型到存储模型的转换，或者说是瞬时状态与持久状态的相互转换。

Redis有两种持久化方式，默认是snapshot方式，实现方法是定时将内存的快照持久化到硬盘，这种方式的缺点是持久化之后如果出现crash则会丢失一段数据。另外一种是aof方式，在写入内存数据的同时将操作命令保存到日志文件中。

**快照方式：**

这种快照方式和虚拟机的快照一样，保存某一时刻的完整数据。Redis在使用这种方式做持久化的时候，定期（默认5分钟）会先写入到一个临时文件，写入完成后，会用这个文件去替换上次的旧的文件。这种方式的好处是，任何一次的快照文件都是完整可用的。但是缺点是，它每隔一段时间（默认最快1分钟，最慢15分钟）做一次，所以会存在一段时间的数据丢失。

**AOF方式：**

这种方式就是把对Redis内存数据的的写指令记录下来，这些指令会被记录在AOF文件的末尾，然后每秒做一次fsync操作（默认每秒一次），把指令在后台在执行一次执行过程其实就是修改磁盘上的数据库内容。所以如果出现故障也只丢失1秒的数据。

上面这种方式就很类似于传统数据库服务器的事务日志。

如果遇到在追加日志的时候遇到意外，可以使用redis-check-aof工具进行日志修复。

因为采用了追加方式，所以AOF会越来越大（这一点又和传统数据库不一样，传统数据库事务日志文件都比较小），因此redis有另外一个机制就是AOF文件重写，当AOF文件达到一个设定的阈值后，会自动启动AOF文件压缩，只保留可以恢复数据的最小指令集。

通过上面的对AOF的描述，可以看到AOF是一个面向过程的，而RDB是面向对象的。

AOF方式的有点：

* 丢失数据最小

AOF方式的缺点：

* 同等数据量，AOF文件比RDB文件体积大
* AOF恢复速度比RDB方式慢

### 6. Redis的线程安全问题

#### 补充：MySQL 事务

* **原子性：**一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。（int i=0; 两个线程同时进行get(i), i++, put(i)这个操作，结果为3正确）
* **一致性：**在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
* **隔离性：**数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
* **持久性：**事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

#### Redis 线程不安全

具体问题实例

有个键，假设名称为myNum，里面保存的是阿拉伯数字，假设现在值为1，存在多个连接对myNum进行操作的情况，这个时候就会有并发的问题。假设有两个连接linkA和linkB，这两个连接都执行下面的操作，取出myNum的值，+1，然后再存回去，看看下面的交互：

linkA get myNum => 1

linkB get myNum => 1

linkA set muNum => 2

linkB set myNum => 2

执行完操作之后，结果可能是2，这和我们预期的3不一致。

要实现线程安全可以加锁