系统安全稳定方案

数据库

- 数据库应用服务器集群

  Oracle的三种高可用集群方案
  RAC（Real Application Clusters）
  多个 Oracle 服务器组成一个共享的 Cache，而这些 Oracle 服务器共享一个基于网络的存储。这个系统可以容忍单机/或是多机失败。不过系统内部的多个节点需要高速网络互连，基本上也就是要全部东西放在在一个机房内，或者说一个数据中心内。如果机房出故障，比如网络不通，那就坏了。所以仅仅用 RAC 还是满足不了一般互联网公司的重要业务的需要，重要业务需要多机房来容忍单个机房的事故。

  Data Guard（最主要的功能是冗灾）
  Data Guard 这个方案就适合多机房的。某机房一个 production 的数据库，另外其他机房部署 standby 的数据库。Standby 数据库分物理的和逻辑的。物理的standby数据库主要用于 production 失败后做切换。而逻辑的 standby 数据库则在平时可以分担 production 数据库的读负载。

  MAA
  MAA(Maximum Availability Architecture)其实不是独立的第三种，而是前面两种的结合，来提供最高的可用性。每个机房内部署 RAC 集群，多个机房间用 Data Guard 同步。

- 数据库读写分离及分库分表

  使用数据库中间件实现读写分离和分库分表

  1、shardingsphere

  2、MyCat

  3、DBProxy

- 数据库备份

  一、导出 / 导入（EXP/IMP）

  （1）、表方式（T方式），将指定表的数据导出。

  （2）、用户方式（U方式），将指定用户的所有对象及数据导出。

  （3）、全库方式（Full方式），瘵数据库中的所有对象导出。

  二、冷备份

  冷备份发生在数据库已经正常关闭的情况下，当正常关闭时会提供给我们一个完整的数据库。冷备份时将关键性文件拷贝到另外的位置的一种说法。对于备份Oracle信息而言，冷备份时最快和最安全的方法。冷备份的优点是：

  1、 是非常快速的备份方法（只需拷文件）

  2、 容易归档（简单拷贝即可）

  3、 容易恢复到某个时间点上（只需将文件再拷贝回去）

  4、 能与归档方法相结合，做数据库“最佳状态”的恢复。

  5、 低度维护，高度安全。

  但冷备份也有如下不足：

  1、 单独使用时，只能提供到“某一时间点上”的恢复。

２、 再实施备份的全过程中，数据库必须要作备份而不能作其他工作。也就是说，在冷备份过程中，数据库必须是关闭状态。

３、 若磁盘空间有限，只能拷贝到磁带等其他外部存储设备上，速度会很慢。

４、 不能按表或按用户恢复。

如果可能的话（主要看效率），应将信息备份到磁盘上，然后启动数据库（使用户可以工作）并将备份的信息拷贝到磁带上（拷贝的同时，数据库也可以工作）。冷备份中必须拷贝的文件包括：

１、 所有数据文件

２、 所有控制文件

３、 所有联机REDO LOG文件

４、 Init.ora文件（可选）

值得注意的使冷备份必须在数据库关闭的情况下进行，当数据库处于打开状态时，执行数据库文件系统备份是无效的。

三、热备份

热备份是在数据库运行的情况下，采用archivelog mode方式备份数据库的方法。所以，如果你有昨天夜里的一个冷备份而且又有今天的热备份文件，在发生问题时，就可以利用这些资料恢复更多的信息。热备份要求数据库在Archivelog方式下操作，并需要大量的档案空间。一旦数据库运行在archivelog状态下，就可以做备份了。热备份的命令文件由三部分组成：

1． 数据文件一个表空间一个表空间的备份。

（1） 设置表空间为备份状态

（2） 备份表空间的数据文件

（3） 回复表空间为正常状态

2． 备份归档log文件

（1） 临时停止归档进程

（2） log下那些在archive rede log目标目录中的文件

（3） 重新启动archive进程

（4） 备份归档的redo log文件

3． 用alter database bachup controlfile命令来备份控制文件

热备份的优点是：

1． 可在表空间或数据库文件级备份，备份的时间短。

2． 备份时数据库仍可使用。

3． 可达到秒级恢复（恢复到某一时间点上）。

4． 可对几乎所有数据库实体做恢复

5． 恢复是快速的，在大多数情况下爱数据库仍工作时恢复。

热备份的不足是：

1． 不能出错，否则后果严重

2． 若热备份不成功，所得结果不可用于时间点的恢复

3． 因难于维护，所以要特别仔细小心，不允许“以失败告终”。

- 数据库监控部署

  1、Oracle数据库常见性能监控指标

  Oracle数据库常见性能指标主要有：当前登录数、非阻塞锁数、当前死锁数、阻塞锁数、当前锁数、会话数等内容

Oracle:orcl_innet.zhetao.com_1521: ORACLE系统性能:OR 曲线图 柱状图 删除所选 常见性能	
当前登录数	42.0
非阻塞锁数	21.0
当前死锁数	0.0
阻塞锁数	0.0
当前锁数	21.0
ora.sessions.act	42.0
ora.process	53.0
ora.bgpprocess	362.0
不活动会话数	0.0
当前会话数	42.0
硬解析次数	2504.0
REDO大小	3.847352E7
物理写次数	10795.0
物理次读	13040.0
队列转换数	92953.0
队列请求数	2953735.0
队列死锁数	0.0
队列等待数	20.0
当前打开的游标数	31.0

2、Oracle数据库的SGA、PGA与Pool监控指标

SGA(System Global Area)：由所有服务进程和后台进程共享；PGA(Program Global Area)：由每个服务进程、后台进程专有；每个进程都有一个PGA。

Oracle:orcl_innet.zhetao.com_1521: ORACLE系统性能:SGA性能: SGA性能	
Redo Buffers	7.488 MB
Redo Buffers KBytes	7668.0 KB
Redo Buffers Bytes	7852032.0 Bytes
Redo Buffers GBytes	0.0070 GB
Database Buffers Bytes	1.996488704E9 Bytes
Database Buffers KBytes	1949696.0 KB
Database Buffers	1904.0 MB
Database Buffers GBytes	1.859 GB
Variable Size Bytes	5.70428296E8 Bytes
Variable Size KBytes	557058.883 KB
Variable Size	544.003 MB
Variable Size GBytes	0.531 GB
Fixed Size GBytes	0.0080 GB
Fixed Size	8.509 MB
Fixed Size KBytes	8713.117 KB
Fixed Size Bytes	**8922232.0 Bytes**
Oracle:orcl_innet.zhetao.com_1521: ORACLE系统性能:ora.sgastat: ora.sgastat	
java pool Free	100.0 %
large pool Usage	1.465 %
large pool Free	98.535 %
shared pool Usage	68.359 %
shared pool Free	31.641 %
java pool Usage	0.0 %

3、Oracle数据库重要资源利用率监控

通过对Oralce数据库的重要资源利用率监控，可以得到数据库的吞吐量、负载等重要性能指标。

Oracle:orcl_innet.zhetao.com_1521: ORACLE系统性能:重要资源使用/占用率: 重要资源使用/占用率	
内存排序占用率	100.0 %
SQL软解析占用率	91.822 %
回滚争用率	0.0070 %
回滚总计	14920.0
回滚等待	1.0
共享区字典缓存区命中率	92.062 %
共享池命中率	93.529 %
共享池命中率(不含reloads)	93.41 %
缓冲区命中率	97.747 %
SQL解析占用总运行时间比	0.225
SQL解析占用CPU	90.433 %
磁盘排度占用率	0.0 %

4、Oracle数据库表空间监控

表空间是用来存放表数据的重要分配空间，一般表空间与数据库文件对应，表空间用完如果没有开启自动扩容，Oracle将产生异常，因此对Oralce数据库的表空间监控是非常重要的监控指标。

Oracle:orcl_innet.zhetao.com_1521: ORACLE表空间性能:SYSAUX: SYSAUX	
SYSAUX总空间(MB)	520.0 MB
SYSAUX空闲空间(MB)	26.12 MB
SYSAUX空闲块(blocks)	3344.0 blocks
SYSAUX已用空间(MB)	493.88 MB
SYSAUX 空闲率	5.02 %
SYSAUX 使用率	94.98 %
Oracle:orcl_innet.zhetao.com_1521: ORACLE表空间性能:SYSTEM: SYSTEM	
SYSTEM总空间(MB)	820.0 MB
SYSTEM空闲空间(MB)	8.88 MB
SYSTEM空闲块(blocks)	1136.0 blocks
SYSTEM已用空间(MB)	811.12 MB
SYSTEM 空闲率	1.08 %
SYSTEM 使用率	98.92 %
Oracle:orcl_innet.zhetao.com_1521: ORACLE表空间性能:UNDOTBS1: UNDOTBS1	
UNDOTBS1总空间(MB)	535.0 MB
UNDOTBS1空闲空间(MB)	500.5 MB
UNDOTBS1空闲块(blocks)	64064.0 blocks
UNDOTBS1已用空间(MB)	34.5 MB
UNDOTBS1 空闲率	93.55 %
UNDOTBS1 使用率	6.45 %
Oracle:orcl_innet.zhetao.com_1521: ORACLE表空间性能:USERS: USERS	
USERS总空间(MB)	5.0 MB
USERS空闲空间(MB)	4.0 MB
USERS空闲块(blocks)	512.0 blocks
USERS已用空间(MB)	1.0 MB
USERS 空闲率	80.0 %
USERS 使用率	20.0 %

- 数据库巡检

  具体请参考<<数据库巡检.docx>>文档

- 数据库表设计

  具体请参考<<58同城mysql军规.md>>文档

- 数据库连接数设置，具体根据服务器的配置来设置

- SQL优化

  1、制定SQL优化目标
  获取待优化SQL、制定优化目标：从AWR、ASH、ORA工具等主动发现有问题的SQL、用户报告有性能问题DBA介入等，通过对SQL的执行情况进行了解，先初步制定SQL的优化目标。
  2、检查执行计划
  explain工具、sql*plus autotrace、dbms_xplan、10046、10053、awrsqrpt.sql等。 执行计划是我们进行SQL优化的核心内容，无计划，不优化。看执行计划有一些技巧，也有很多方式，各种方式之间是有区别的。
  3、检查统计信息
  ORACLE使用DBMS_STATS包对统计信息进行管理，涉及系统统计信息、表、列、索引、分区等对象的统计信息，统计信息是SQL能够使用正确执行计划的保证。我们知道，ORACLE CBO优化器是利用统计信息来判断正确的执行路径，JOIN方式的，因此，准确的统计信息是产生正确执行计划的首要条件。

可以从这个图看出，一条SQL产生执行计划需要经过哪些步骤，在我看来：1、正确的查询转换；2、准确的统计信息，是产生正确执行计划的重要保证。当然，还有BUG，或优化器限制等也会导致SQL效率低下，无法产生正确的执行计划。

4、检查高效访问结构

重要的访问结构，诸如索引、分区等能够快速提高SQL执行效率。表存储的数据本身，如碎片过多、数据倾斜严重、数据存储离散度大，也会影响效率。

5、检查影响优化器的参数

optimizer_mode、optimizer_index_cost_adj、optimizer_dynamic sampling、*optimizer_mjc_enabled*、optimizer_cost_based_transformation、hash_join_enable等对SQL执行计划影响较大。比如有时候我们通过禁用_optimizer_mjc_enabled 参数，让执行计划不要使用笛卡尔积来提升效率，因为这个参数开启有很多问题，所以一般生产库都要求禁用。

6、SQL语句编写问题

SQL语句结构复杂、使用了不合理的语法，比如UNION代替UNION ALL都可能导致性能低下。 并不是说ORACLE优化器很强大了，我们就可以随便写SQL了，那是不正确的。SQL是一门编程语言，它能够执行的快，是有一些普遍的规则的，遵循这种编程语言特性，简化语句，才能写出好的程序。SQL语句编写出了问题，我们就需要改写，就需要调整业务，改涉及等。

7、SQL优化器限制导致的执行计划差

这个很重要，统计信息准确，SQL也不复杂，索引也有...都满足，为什么我的SQL还是差，那么得考虑优化器限制因素了。这里说1点常见的执行计划限制，当semi join与or连用的时候（也就是exists(subquery) or ...或者in (subquery) or...，如果执行计划中因为OR导致有FILTER操作符，就得注意了，可能慢的因素就和OR有关。这时候我们得改写SQL，当然改写为UNION或UNION ALL了。

OK，以上全部检查完毕，我的系统还是很差，功能还是很慢，或者已经无法从SQL本身进行调整提升性能了，那咋办？优化设计，这是终极方法。有些东西不优化设计是无法解决的，比如业务高峰期跑了一堆SQL，CPU已经很吃紧，又不给增加机器，突然上线一个耗资源的业务，其他SQL已无法调整。那只能优化设计，比如有些耗资源的业务可以换时间段执行等。

以上几点，是我们进行优化需要考虑的地方，可以逐步检查。当然，80%到90%的纯SQL性能调整，我们通过建立索引，收集正确统计信息，改写避免优化器限制，已经能够解决了。

- 数据库告警设定通知规则

  1、短信

  2、邮件

  3、钉钉

  4、微信

- 数据库表敏感信息需要脱敏和匿名化

中间件

redis

一、键值设计

1. key名设计

- (1)【建议】: 可读性和可管理性

以业务名(或数据库名)为前缀(防止key冲突)，用冒号分隔，比如业务名:表名:id

```
ugc:video:1
```

- (2)【建议】：简洁性

保证语义的前提下，控制key的长度，当key较多时，内存占用也不容忽视，例如：

```
user:{uid}:friends:messages:{mid}简化为u:{uid}:fr:m:{mid}。
```

- (3)【强制】：不要包含特殊字符

反例：包含空格、换行、单双引号以及其他转义字符

详细解析

2. value设计

- (1)【强制】：拒绝bigkey(防止网卡流量、慢查询)

string类型控制在10KB以内，hash、list、set、zset元素个数不要超过5000。

反例：一个包含200万个元素的list。

非字符串的bigkey，不要使用del删除，使用hscan、sscan、zscan方式渐进式删除，同时要注意防止bigkey过期时间自动删除问题(例如一个200万的zset设置1小时过期，会触发del操作，造成阻塞，而且该操作不会不出现在慢查询中(latency可查))，查找方法和删除方法

详细解析

- (2)【推荐】：选择适合的数据类型。

例如：实体类型(要合理控制和使用数据结构内存编码优化配置,例如ziplist，但也要注意节省内存和性能之间的平衡)

反例：

```
set user:1:name tom
set user:1:age 19
set user:1:favor football
```

正例:

```
hmset user:1 name tom age 19 favor football
```

3.【推荐】：控制key的生命周期，redis不是垃圾桶。

建议使用expire设置过期时间(条件允许可以打散过期时间，防止集中过期)，不过期的数据重点关注idletime。

二、命令使用

1.【推荐】 O(N)命令关注N的数量

例如hgetall、lrange、smembers、zrange、sinter等并非不能使用，但是需要明确N的值。有遍历的需求可以使用hscan、sscan、zscan代替。

2.【推荐】：禁用命令

禁止线上使用keys、flushall、flushdb等，通过redis的rename机制禁掉命令，或者使用scan的方式渐进式处理。

3.【推荐】合理使用select

redis的多数据库较弱，使用数字进行区分，很多客户端支持较差，同时多业务用多数据库实际还是单线程处理，会有干扰。

4.【推荐】使用批量操作提高效率

```
原生命令：例如mget、mset。
非原生命令：可以使用pipeline提高效率。
```

但要注意控制一次批量操作的**元素个数**(例如500以内，实际也和元素字节数有关)。

注意两者不同：

```
1. 原生是原子操作，pipeline是非原子操作。
2. pipeline可以打包不同的命令，原生做不到
3. pipeline需要客户端和服务端同时支持。
```

5.【建议】Redis事务功能较弱，不建议过多使用

Redis的事务功能较弱(不支持回滚)，而且集群版本(自研和官方)要求一次事务操作的key必须在一个slot上(可以使用hashtag功能解决)

6.【建议】Redis集群版本在使用Lua上有特殊要求：

- 1.所有key都应该由 KEYS 数组来传递，redis.call/pcall 里面调用的redis命令，key的位置，必须是KEYS array, 否则直接返回error，"-ERR bad lua script for redis cluster, all the keys that the script uses should be passed using the KEYS array"
- 2.所有key，必须在1个slot上，否则直接返回error, "-ERR eval/evalsha command keys must in same slot"

7.【建议】必要情况下使用monitor命令时，要注意不要长时间使用。

三、客户端使用

1.【推荐】

避免多个应用使用一个Redis实例

正例：不相干的业务拆分，公共数据做服务化。

2.【推荐】

使用带有连接池的数据库，可以有效控制连接，同时提高效率，标准使用方式：

```
执行命令如下：
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    //具体的命令
    jedis.executeCommand()
} catch (Exception e) {
    logger.error("op key {} error: " + e.getMessage(), key, e);
} finally {
    //注意这里不是关闭连接，在JedisPool模式下，Jedis会被归还给资源池。
    if (jedis != null)
        jedis.close();
}
```

下面是JedisPool优化方法的文章:

- Jedis常见异常汇总
- JedisPool资源池优化

3.【建议】

高并发下建议客户端添加熔断功能(例如netflix hystrix)

4.【推荐】

设置合理的密码，如有必要可以使用SSL加密访问（阿里云Redis支持）

5.【建议】

根据自身业务类型，选好maxmemory-policy(最大内存淘汰策略)，设置好过期时间。

默认策略是volatile-lru，即超过最大内存后，在过期键中使用lru算法进行key的剔除，保证不过期数据不被删除，但是可能会出现OOM问题。

其他策略如下：

- allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。
- allkeys-random：随机删除所有键，直到腾出足够空间为止。
- volatile-random:随机删除过期键，直到腾出足够空间为止。
- volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。
- noeviction：不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。

四、相关工具

1.【推荐】：数据同步

redis间数据同步可以使用：redis-port

2.【推荐】：big key搜索

redis大key搜索工具

3.【推荐】：热点key寻找(内部实现使用monitor，所以建议短时间使用)

facebook的redis-faina

> 阿里云Redis已经在内核层面解决热点key问题，欢迎使用。

五 附录：删除bigkey

> 1. 下面操作可以使用pipeline加速。
> 2. redis 4.0已经支持key的异步删除，欢迎使用。

1. Hash删除: hscan + hdel

```
public void delBigHash(String host, int port, String password, String bigHashKey)
{
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<Entry<String, String>> scanResult = jedis.hscan(bigHashKey,
cursor, scanParams);
        List<Entry<String, String>> entryList = scanResult.getResult();
        if (entryList != null && !entryList.isEmpty()) {
            for (Entry<String, String> entry : entryList) {
                jedis.hdel(bigHashKey, entry.getKey());
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    //删除bigkey
    jedis.del(bigHashKey);
}
```

2. List删除: ltrim

```
public void delBigList(String host, int port, String password, String bigListKey)
{
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    long llen = jedis.llen(bigListKey);
    int counter = 0;
    int left = 100;
    while (counter < llen) {
```

```
        //每次从左侧截掉100个
        jedis.ltrim(bigListKey, left, llen);
        counter += left;
    }
    //最终删除key
    jedis.del(bigListKey);
}
```

3. Set删除: sscan + srem

```
public void delBigSet(String host, int port, String password, String bigSetKey)
{
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<String> scanResult = jedis.sscan(bigSetKey, cursor,
scanParams);
        List<String> memberList = scanResult.getResult();
        if (memberList != null && !memberList.isEmpty()) {
            for (String member : memberList) {
                jedis.srem(bigSetKey, member);
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    //删除bigkey
    jedis.del(bigSetKey);
}
```

4. SortedSet删除: zscan + zrem

```
public void delBigZset(String host, int port, String password, String bigZsetKey)
{
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<Tuple> scanResult = jedis.zscan(bigZsetKey, cursor,
scanParams);
        List<Tuple> tupleList = scanResult.getResult();
        if (tupleList != null && !tupleList.isEmpty()) {
            for (Tuple tuple : tupleList) {
                jedis.zrem(bigZsetKey, tuple.getElement());
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    //删除bigkey
```

```
    jedis.del(bigZsetKey);
}
```

六 redis高可用

1、redis搭建主从同步方案

2、禁止外网访问 Redis

3、根据业务的需要设置好redis的备份策略

RDB持久化和AOF持久化
RDB持久化是一种生成「快照」数据的方式，它会根据配置文件（redis.conf）中的持久化策略在合适的时机自动去dump整个Redis服务器在「某个时刻」的中的全量内存数据，即某个时刻的快照数据。并将快照数据保存在一个名叫dump.rdb的文件中，这些快照数据以二进制格式压缩存储。
RDB持久化方式关注点在于快照数据，每次触发RDB持久化都是全量保存某个时间点上的所有内存数据。就这一点而言，它很适合备份场景，用于灾难恢复。它有如下优点：
RDB持久化生成的dump.rdb文件是一个经过压缩的紧凑的二进制文件，加载/恢复速度很快。
RDB持久化也有缺点：
没法做到实时/秒级持久化，因为每次RDB持久化都会fork一个子进程来生成快照数据，fork属于重量级操作，频繁fork会让cpu和内存吃不消，影响Redis性能。

AOF持久化保存的是一种逻辑日志，即记录的是一条条写操作的命令，而不是像RDB持久化那样记录物理数据。它在恢复数据的时候，是直接执行AOF文件中的一条条Redis命令来重建整个数据集的。
AOF持久化的优点：
能够做到实时/秒级别的持久化，数据的实时性更好。
AOF持久化优点：
AOF文件体积会比RDB大，如果数据集很大，AOF重写和AOF文件加载/恢复都将是一个很耗资源和耗时的操作。

Redis的两种持久化方式各有特色，我们生产环境一般不会只用其中一种，而是同时使用两种。
例如RDB可以结合cron定时任务去定期生成备份数据，用于灾难恢复；同时，AOF因为支持实时持久化，它记录的数据集是最实时的，所以我们也会同时开启AOF持久化，应对一些对数据实时完整性要求较高的场景。但是AOF也可能会损坏无法修复，所以两种方式并用对数据才是最安全的。

4、提前计算出系统可能会用的内存大小，合理的分配内存。需要注意在开启持久化模式下，需要预留更多的内存提供给Fock的子进程做数据磁盘flush操作

5、避免生产环境使用keys命令，因为redis是单线程，执行keys操作会卡顿业务

6、为Redis添加密码验证

修改redis.conf文件，添加

```
requirepass mypassword
```

7、redis最大连接数配置

```
connected_clients
```

8、设置timeout和tcp-keepalive来清理失效的连接

9、设置合理的内存回收策略，保证内存可用性的同时能适当的提供缓存的命中率

- kafka

- elasticsearch

应用(服务)设计与部署

- 应用或服务部署在linux服务器上，设置好服务器的最大连接数

```
# vi /etc/security/limits.conf

*       soft    nofile  65535
*       hard    nofile  65535
*       soft    nproc  65535
*       hard    nproc  65535
```

- 应用或服务模块拆分
- 核心应用或服务集群，支持横向扩展实现负载均衡
- 核心应用或服务高可用，防止单点故障
- 应用或服务治理/监控(服务QPS预警和限制)，可以在服务能力即将达到临界时，向运维人员发送告警提醒，在实际达到临界时，拒绝更多请求压垮服务；
- 对接接口统一走网关来控制访问(调用次数、黑白名单)/降级/限流/熔断
- 某些业务在允许的情况下异步化(mq等)
- 兜底/容错，对于某些业务我们都必须有一个兜底的预案
- 核心业务，比如交易业务端要实现幂等设计
- 核心接口及方法请求的入参和出参需要记录详细的日志，便于定位问题
- 后端防止sql注入
- 应用或服务中正确使用数据库连接池，连接池大小一般设置的公式 连接数=((核心数 * 2) + 有效磁盘数)
- 应用或服务的配置文件比如数据库需要使用对称加密配置
- 对于第三方调用接口后端需要验证签名sign，像分页功能建议限制最大记录数范围值
- 对于热点数据使用redis缓存，使用redis客户端连接池，设置好最大连接数
- 应用使用时尽量为每块缓存设置时效性，避免冷数据长时间占用资源

前端安全

- 接口https证书加密传输
- 本地加密混淆
- 授权机制token，获取接口token令牌需要拿appid、timestamp和sign来换，sign=加密(timestamp+key)请求头使用token访问，token设置超时时间，用户每次访问必须检验token的合法性，下发的token建议使用uuid无法猜测
- 防御DOS攻击，用户每次请求带上时间戳timestamp，服务端收到timestamp后跟当前时间进行比对，如果时间差大于一定时间(比如3分钟)，则认为请求失效
- 接口签名sign，防止数据篡改。将token和时间戳加上其他请求参数再用MD5或SHA-1算法（可根据情况加点盐）加密，加密后的数据就是请求sign，服务端接收到请求后以同样的算法签名，并跟当前的签名进行比对，如果不一样，说明参数被更改过，直接返回错误标识。

- 接口权限校验，某些安全性要求高的接口可用rsa等公私钥进行数据加密，对访问API的用户权限进行控制，特殊的请求需要限制，比如分页接口限制最大记录数
- 接口一致性保证
- 接口限流和降级，接口ip地址限制，调用次数和频率的限制
- 接口熔断
- 接口日志记录，接口请求的入参和出参需要记录详细的日志信息(请求ip+参数等)
- 接口防跨步骤调用，
- 核心接口，比如交易、支付等使用短信验证码方式
- 防SQL注入、XSS、CSRF攻击
- 前后端分离项目，接口暴露在上生产前一定要记得关闭访问权限