容错的目标是降低或者最小化故障对系统可用性、可靠性、安全性、持续性等得影响。在软件容错中，常常使用 `fault` (缺陷), `error` (错误), `failure` (故障)来表示系统异常的由来。系统 `缺陷` 在某种特定环境下被激活，到至系统产生 `错误` ，系统错误运行是的是的系统发生某种 `故障` 。

常见方法

容错按系统级别划分，分为三个级别，硬件容错、软件容错以及系统容错。硬件容错常用的方法包括使用冗余、多备份技术、增加内存、能源系统冗余等。硬件错误通常能够够在两个物理机上进行隔离处理。软件容错主要是正对软件的鲁棒性特征进行增强。常见的方法有checkpoint/restart，recovery blocks，N-Version Programs等。对于系统容错，设计一个独立与目标系统的子系统，通过定义定义规则来容忍系统 `缺陷` 。对 `缺陷` 的处理，有以下几类技术：

1. 使用 `缺陷避免` 技术来避一些错误。使用成熟的设计方法论、验证以及确认方法论、代码检查、上线前的演练等。

2. 在可能会存在的 `缺陷` 时，可以选择 `缺陷移除` 技术，例如测试、集成测试、回归测试、背靠背测试等；

3. 或者是在遭遇错误是， `缺陷回避` 的方式，是的潜在的 `缺陷` 不会被激活。常见技术是通过重新配置系统来达到避免的目标；

4. `缺陷容忍` 技术，系统能够对 `缺陷` 进行侦测、诊断、孤立、覆盖、不错、以及系统恢复。使用以上多种技术混合。

常见的机遇鲁棒性的方法

1. Process Pairs

Pocess Pairs：保证系统在某一个时候总能有一个进程来处理客户的输入请求。他能处理短暂的软件错误。

框架代码如下：

```
    //create shadow process; primary return
    int ft = backup();//server process.
    ...
    forever{
        wait_for_request(Request);
        process_request(Request);
    }
```

`backup()` 实现如下:

```
int backup(){
    int ret, restarts = 0;//count number of child procs
    for(;;restarts++){
        ret = fork();//clone process
        if(ret == 0){//child?
            return restarts;
        }
        while(ret != waitpid(ret,0,0))
            ;//parent waits for child to terminate
    }
}
```

2. Graceful Degradation

在系统遭遇某个错误不能提供完整功能，系统可以降低自己服务能力。

上述 `backup()` 方法中 `while` 循环中 `waitpid` 需要等待到child创建成功。可能永远不会成功，需要改进。

`backup()` 实现如下:代码增加了line 6,7

```
1  int backup(){
2      int ret, restarts = 0;//count number of child procs
3      for(;;restarts++){
4          ret = fork();//clone process
5          if(ret < 0 ){
6              log("backup:    ....");
7              return -1;//即便fork不成功，也可以返回
8          }
9          if(ret == 0){//child?
10             return restarts;
11         }
12         while(ret != waitpid(ret,0,0))
13             ;//parent waits for child to terminate
14     }
15 }
```

3. Selective Retry

重复调用仅当有机会重试成功。例如，内存短缺消失；主要针对突然地高负载资源短缺。它能够增加资源分配成功的可能性

代码如下：

```
//create backup process; primary return
int ft = backup();//server process.
...
forever{
    //可能成功，如果条件应许。不成功，继续forever
    if(ft < 0 ){ ft = backup();}//可能死循环；状态一直打断。
    wait_for_request(Request);
    process_request(Request);
}
```

4. State Handling

在系统不能提供服务后，又要保证client的无状态属性。服务端需要持续保存当前的状态，用于故障后的重试。

```
//create backup process; primary return
int ft = backup();//server process.
...
forever{
    wait_for_request(Request);
    get_session_state(Request);
    if(num_retries < N){//定义重复次数
        process_request(Request);
        store_session_state(Request);
    }else{
        return_error();
    }

}
```

5. Linking Process

有些程序进程见是相互以来的，如果某个进程出错，其他以来的进程需要侦测到错误，明确做相应的处理，通常为结束全部依赖进程。

```
int fd[2];//pipe fd 管道技术
int backup(){
    ...
    pipe(fd);
    ret = fork();
    if(ret == 0){
        close(fd[1]);//写端
        return restarts++;
    }//parent closes other end
    close(fd[0]);//读端
}
```

6. Rejuvenation

Rejuvenation常用于不可重现得一些问题，比如资源耗尽，线程间的不同调度等。通常是使用以下几类，1. 周期性，如每周星期天4am；2.基于负载，如没10000个请求后；3. 预测式，在线监控系统可能预测之后某个时候可能会有crash，提前干预。

```
int ft = backup();//server process.
...
for(i = 0; i < N || ft <0; ++i){
    if(ft < 0 ){
        ft = backup();
        if ( ft >= 0 ) i = 0;//reset index
    }
    wait_for_request(Request);
    process_request(Request);
}
```

7. Checkpointing

周期性的保存进程的状态。如果需要crash错误，回滚到最近保存的状态。

代码如下：

```
    pid grandparent = getpid();//after two forks
    ...
    //Simple Verision: for N requests
    for(int nxt_ckpt = 0;;nxt_ckpt--){
        if(nxt_ckpt <= 0){
            pid parent = getpid();
            if(backup() >= 0 && grandparent != parent){//fork sucessful and not first fork
                kill(grandparent,KILL);
                grandparent = parent;
                nxt_ckpt = N;
            }
        }
        wait_for_request(Request);
        process_request(Request);
    }
```

`backup()` 同上：

```
    int backup(){
        int ret, restarts = 0;//count number of child procs
        for(;;restarts++){
            ret = fork();//clone process
            if(ret < 0 ){
                log("backup:    ....");
                return -1;//即便fork不成功，也可以返回
            }
            if(ret == 0){//child?
                return restarts;
            }
            while(ret != waitpid(ret,0,0))//死循环
                ;//fork新的child如果当前child已经crashed了。
        }
    }
```

8. Update Lost

在两个checkpoint点之间系统故障，需要保存客户请求。在rollback的时候重新处理这些请求。

```
request_no = 0;
...
for(int nxt_ckpt = 0;;nxt_ckpt--){
    //周期校检,如果校检点不是最新状态
    if(checkpoint(&nxt_ckpt) == RECOVERY){
        //处理log
        while((request_no+1,R) in log){
            process_request(R);
            request_no++;
        }
    }
    wait_for_request(Request);
    log_to_disk(++request_no,Request);//保存所有请求
    pocess_request(Request);
}
```

9. Application State Scrubbing

系统状态的保证可以通过checkpoint以及logging的方式，但是在对特定的运用是，就需要应用自身能保存并回滚状态。

代码如下：

```
int ft = backup();
restore_application_state();//child读最新的校检点
...
for(int nxt_scrub = N;;nxt_scrub--){
    //周期处理 after N request
    ...
    //only if child process
    if(nxt_scrub <= 0 && ft >= 0){
        if(save_application_state() >= 0){
            exit(0);
        }
    }
    wait_for_request(Request);
    pocess_request(Request);
}
```

10. Process Pools

主要针对资源漏洞的问题,在N个请求后，结束**进程**。操作系统释放所有进程资源。或者使用资源预分配技术，按照经验设定好某些请求资源的需求量，当侦测到程序需要求是，分配资源。

11. Recovery Block

框架结构如下：

```
ensure { postcondition} by
    {primary alternative}
    else by {2nd alternative}
    else by {3rd alternative}
    ...
    else by {final alternative}
    else error;
```

比如一个处理数组排序：

```
ensure { A.sort()} by //检查A是否已经排序完成
    {A.my_new_quick_sort();}
    else by {A.simple_bubble_sort();}//老式但是鲁棒性的方法
    else throw exception;//如果所有方法都不满足，抛出异常
```

代码实现如下：

```
forever{
    if(! alternative_left()){
        throw exception;
    }
    /*
    生成一个child执行下一个替代方法。
    */
    backup(parent);
    set_watchdog(timeout);//watchdog处理性能故障
    execute next alternative;

    /*
    替代方法执行成功，满足判断条件，结束父进程，返回
    */
    if(postcondition()){
        reset_watchdog();//成功后重置watchdog
        parent.terminate();
        return ;
    }
    exit(0)
}
```

12. Microreboot

通过解耦系统组件，使得系统在遭遇故障时，组需要重启需要的组件，而不必重启整个系统。

核心是组件和数据分离。数据的处理通过持久化存储的方式保证一致。

13. State Correction